CANADIAN THESES
THÈSES CANADIENNES

QUNJIE, WANG
NETWORKED VIRTUAL REALITY.

ISBN
0-315-95099-4

National Library of Canada
Acquisitions and Bibliographic Services Branch
395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada
Direction des acquisitions et des services bibliographiques
395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

UNIVERSITY OF ALBERTA

Networked Virtual Reality

BY

Qunjie Wang

©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

DEPARTMENT OF COMPUTING SCIENCE

Edmonton, Alberta
Fall 1994

National Library of Canada
Acquisitions and Bibliographic Services Branch
395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada
Direction des acquisitions et des services bibliographiques
395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file *Votre référence*

Our file *Notre référence*

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-95099-4

Canada

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: **Qunjie Wang**

TITLE OF THESIS: **Networked Virtual Reality**

DEGREE: **Master of Science**

YEAR THIS DEGREE GRANTED: **1994**

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as hereinbefore provided neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

(Signed) Wang Qunjie

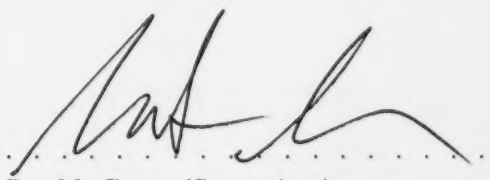
Qunjie Wang
Building 2, Room 507
Yudao Street No.30
Nanjing, Jiangsu
P.R.China 210016

Date: Aug. 24. 1994

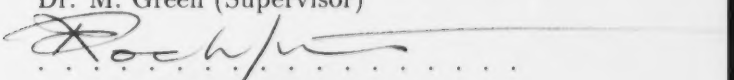
UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

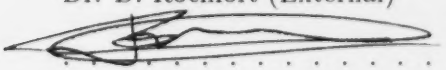
The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Networked Virtual Reality** submitted by *Qunjie Wang* in partial fulfillment of the requirements for the degree of *Master of Science*



Dr. M. Green (Supervisor)



Dr. D. Rochfort (External)



Dr. J. Buchanan (Examiner)

Dr. P. Rudnicki (Chair)

Date: Aug. 18, 1994

DEDICATED

To My Parents,
My Sister and My Grandmother

Abstract

A *Virtual Reality* world or a *Virtual Environment* is a highly interactive simulation of a real or imaginary world. It is one form of a Highly Interactive Three Dimensional User Interface. Now that many single-user applications have been successfully implemented, a hotter topic is *Networked Virtual Reality*, where multiple virtual environments (multiple users) at different geographical locations, share information over a network.

This thesis focuses on building an *Environment Manager* (EM) on top of the Object Modeling Language and the Minimal Reality Toolkit. The purpose of EM is to facilitate the creation of multi-user virtual environments that operate on workstations connected over a network; and to allow one or several users to interact with 3D objects in a virtual world, share information and cooperate with each other.

As a new component of the virtual reality software package developed at the University of Alberta's Graphics Laboratory, EM simplifies the task of building a networked virtual world by providing the standard facilities required by a wide range of networked virtual environments.

Acknowledgments

First of all, I am very grateful to my supervisor, Mark Green, whose insight, guidance, encouragement and humor were invaluable to me. I feel very lucky to have had the opportunity to be associated with him over this time, which now seems too brief. I am indebted to him for his immeasurable help and financial assistance; and for suffering through the many drafts of this thesis, and tirelessly telling me to be confident.

I would also like to thank the other members of my committee: John Buchanan and Rochfort Desmond for their evaluation of this thesis and helpful comments. Anne Nield deserves many thanks for being so "picky" at my last thesis draft.

My colleagues at the Graphics Lab deserve a special note of recognition. Their helpful suggestions and patience with my countless questions are greatly appreciated. In particular, Chris Shaw, as a colleague and a friend, helped me overcome many problems. His advice contributed to this thesis work substantially, and I shall definitely miss our fruitful discussions.

I dedicate this thesis to my parents, sister and grandmother, for their love and support, their constant pride in me, and for staying so close to me.

I wish to thank Xiaolin Chen, Li Yao, Magaret Benke, Helen Arnott and Ed Arnott who enriched my life at the University of Alberta. I am very lucky to have known them. Many thanks to Donna Kuhn for her understanding, for loving me like a sister and reading through my first thesis draft.

Special thanks to Yong Zhuang, whose love, understanding and encouragement I ever treasure.

Contents

1 Introduction **1**

1.1 What Is Networked Virtual Reality? 1

1.2 Thesis Motivations 2

1.2.1 The Need For Distribution (Networked Virtual Reality) 2

1.2.2 The Need For Developing A Tool Package 3

1.3 Thesis Goals 5

1.4 Thesis Overview 5

2 Existing Networked VR Systems And Tools **7**

2.1 Centralized Or Replicated Architecture 7

2.2 Different And Identical Local Databases 9

2.3 Case Studies 10

2.3.1 VR-DECK 10

2.3.2 SIMNET, NPSNET And VERN 11

2.3.3 DIVE 13

2.3.4 BrickNet 14

3 Background **16**

3.1 Minimal Reality Toolkit (MR) 16

3.2 Object Modeling Language 17

3.3 Development Of Single User VR Using OML And MR 20

3.4 The MR Toolkit Peer Package 22

3.5 The User Interfaces 23

4 Building The Environment Manager(EM) 24

4.1 Introduction 24

4.2 Networked VR System Architecture 27

4.2.1 Replicated Application Architecture 27

4.2.2 Configuration Of Object Space 29

4.2.3 Architecture And Components Of The Environment Manager 31

4.3 Concurrency Control 44

4.3.1 Issues And Other Approaches 44

4.3.2 Concurrency Control In EM 47

4.4 Unique Instances 51

4.5 Reducing Bandwidth 52

4.6 Shared Information 54

4.6.1 Shared Instance Variables 54

4.6.2 Shared Instances 55

4.6.3 Shared Objects 55

4.6.4 Shared Object Behaviors 58

5 Demonstrations 59

5.1 Skeet Shooting Game 59

5.1.1 Scenario 59

5.1.2 Implementation 59

5.2 Tank Battle 62

5.2.1 Scenario 62

5.2.2 Implementation 62

5.3 The East Edmonton Mall Design Environment 66

5.3.1 Scenario 66

5.3.2 Implementation 67

6 Conclusions And Future Work 71

6.1 Conclusions 71

6.2 Comparison With Other Systems 72
6.3 Discussions And Future Work 74
6.3.1 Improving Reliability 74
6.3.2 Improving Concurrency Control 76
6.3.3 Extensions For Implementing Spatial Locality 78

Bibliography **81**

A Environment Manager(EM) - User Manual **85**
A.1 Invoking The Environment Manager 85
A.2 The EM Description File Grammar 86

B An EM Description File For A Skeet Shooting Game **92**

C An EM Description File For A Tank Battle **94**

D EM Description Files For The East Edmonton Mall Design Environment **97**
D.1 An EM Description File For The Mall Design Studio 97
D.2 An EM Description File For The Plane Museum Environment 98
D.3 An EM Description File For The Statues Studio Environment 99

List of Figures

4.1 EM, OML, MR And Their Relationships 26
4.2 Application Replication Architecture 28
4.3 Networked Configuration Of The Tank Simulation 30
4.4 Networked Configuration Of The Design Studios 32
4.5 Architecture Of The Environment Manager 38

5.1 A Snapshot Of The Multi-player Skeet Shooting Game 61
5.2 A Snapshot Of The Multi-player Tank Battle Simulation 65
5.3 A Snapshot Of The Plane Museum Museum Environment 69
5.4 A Snapshot Of The Statues Design Studio Environment 69
5.5 A Topview Of The Mall Design Studio Environment 70

6.1 A Caching Volume 79

Chapter 1

Introduction

1.1 What Is Networked Virtual Reality?

Throughout the 1980's, user interface design has been dominated by the desk top metaphor mimicking paper documents. While the desk top metaphor is well suited to certain types of applications, there are a number of applications where this metaphor is not appropriate. Most of them consist of three dimensional (3D) environments that the user wants to manipulate in some way. A new metaphor, called Highly Interactive Three Dimensional (HITD) User Interface, was developed for making use of three dimensional displays and interaction techniques. A HITD user interface addresses a 3D application domain, presents the application data in a 3D format and allows a user interface to simulate a virtual world or some aspects of reality. The emphasis is on the construction of a three dimensional world that the user can explore and interact with.

Virtual Reality (VR) or **Virtual Environments** is *one form of HITD user interface.* The main point is to allow natural 3D interaction in an environment that both the user and the application reside in. The user interacts with 3D input devices, and 3D output devices are used to view the application data. The challenge to computer graphics is to make the computer generated world look real, and the objects in the world act realistically. Indeed, in the ultimate display, one will not look at a world through a window, but will be immersed in it, and will change viewpoints

by natural motions of head and body. One will interact directly and naturally with the objects in the world, hearing and feeling them, as well as seeing them.

Virtual Reality technology has been successfully applied to a variety of areas such as scientific/information visualization, entertainment and recreation, art design, training, computer-aided design, telecommunications and programs for the handicapped.

Networked Virtual Reality refers to network-based virtual environments where multiple users, or multiple virtual environments at different geographical locations connected by a network, can share information with each other.

1.2 Thesis Motivations

With many single-user applications being successfully implemented, Networked Virtual Reality is now becoming a hot topic and the need to distribute and scale up single user virtual worlds is becoming apparent. The development effort required to produce a networked virtual environment is quite large, we need to design tools to simplify this development. These are the motivations for this thesis work.

1.2.1 The Need For Distribution (Networked Virtual Reality)

Data communications, including local area networks and wide area networks, move information around the world and create a global information system. The complexity of emerging the system - its dynamic, abstract, and global characteristics, is where the need for *Networked Virtual Reality* enters the story.

"Teleconferencing with realistic sensation" [TK92] aims at allowing smooth and intricate communications among users by reproducing various aspects of face to face conferencing. Networked VR is definitely an appealing method to achieve this goal.

A major difficulty with communications across cultures, whether professional or national, is that concepts can not be well communicated just by simple language

translation. It is especially insufficient when the communicators come from highly specialized knowledge fields, or from national cultures with long divergent histories. Networked VR has the potential for facilitating communication by immersing the communicators within multiple visual representations of the concepts, which are understandable across cultures. The immersion also provides control over those representations.

The eventual merging of several interactive drama projects in Japan and the infamous rock band's desire of interactive concerts where "the audience could participate in creating the performance" [Loe93] indicate the advancing edge of drama and music, ultimately to be accessed through networked VR – networked immersion environments. The STUDIO project, developed at Carnegie Mellon University has designed and constructed a multi-national art museum in networked VR [Loe93].

The need for networked VR in military has already been well demonstrated in applications such as flight simulators and tank combat simulations [All91]. Above all, due to today's information system being globalized and connected by networks, as soon as developers attempt to build virtual worlds of real-world exactness, the need to distribute and scale up the single user virtual world becomes apparent.

In spite of the continuing necessity for networked virtual reality, it actually already exists in a variety of different forms. Networked VR constitutes a small industry, but one with the promise of great potential and growth. A VR tool package has been developed in the Graphics Lab, University of Alberta to provide support for the creation of single user virtual environments [GSW93, Gre94]. This thesis work is geared towards adding a new component to the VR tool package so that it can enable users to develop networked virtual reality easily and quickly.

1.2.2 The Need For Developing A Tool Package

Virtual Reality is immersion based, which means the user will use non-traditional devices such as head-mounted displays (HMDs) and hand directed input devices. A wide range of software development tools, such as X and Motif, have been devel-

oped for the more traditional Windows, Icons, Menus and Pointing (WIWP) style of user interface. These tools significantly simplify WIWP user interface development. Building VR user interfaces requires a similar set of software development tools. The Minimal Reality (MR) Toolkit has been developed in the Graphics Laboratory, University of Alberta for the following reasons [SGLS93]:

- The production of good VR worlds, whether single user or networked, requires a considerable amount of time and programming. A tool can reduce the amount of development work and allows the scientist to tackle a wider range of applications.
- As with the high level tools such as UIMSs, high level tools for VR user interfaces will likely rely on toolkits to provide low level support.
- A Toolkit allows the easy integration of new devices into existing applications to evaluate their effectiveness.
- A framework is needed for developing new interaction techniques, for integrating them with existing applications without too much extra work and for evaluating several competing interaction techniques in the same environment.

The MR Toolkit simplifies the production of VR applications by providing standard hardware facilities required by a wide range of VR user interfaces.

The MR Toolkit peer package is an extension to the MR Toolkit. It provides connection level facilities for transmitting device data and application-specific shared data between MR master processes [SG93a].

The Object Modeling Language(OML) [Gre94] is developed to describe the geometries and behaviors of three dimensional objects used in virtual worlds.

Despite all the available toolkits mentioned above, building a VR system still takes a considerable amount of effort. This thesis work aims to develop a higher level tool called an **Environment Manager**, on top of our existing toolkit, to relieve the effort of initializing and running virtual environments from the programmer.

1.3 Thesis Goals

In this thesis, we present an environment manager that can run both single user and networked virtual environments consisting of OML objects and instances.

The choice of running a *single* or a *networked* VR world is specified in the command line. The environment manager allows the construction of virtual environments without writing any C or MR code. Only a script file is needed to generate the environment. The script is a procedural language defined by EM.

In the single user mode, the script file describes the device configuration, OML object files and information about parameters and behaviors for defined OML objects and instances.

For networked virtual environments, in addition to the information mentioned for building a single user VR world, information about concurrency control schemes, objects and instances that are shared with other users and shared variables in these instances are also specified in the script file.

The environment manager uses a description file to construct the initial state of a virtual environment and produces instance behaviors while the environment is running.

Mainly, we design and implement:

- The networked virtual reality system architecture; concurrency control schemes; interaction techniques, and communication protocols for shared information including: variables, instances and objects.

- A two-pass EM description file parser to parse the description file when EM starts the virtual world and to parse the sub-EM description file code that EM receives at run-time.

1.4 Thesis Overview

This thesis starts with an overview of previously developed networked virtual

environments and toolkits, emphasizing their application semantics and system architecture as well as communication and control techniques. Next, a brief description of the existing tools, such as the Minimal Reality (MR) Toolkit, MR Toolkit peer package and the Object Modeling Language (OML), is presented. We then describe several design issues and the implementation of the Environment Manager (EM) for both single user and networked virtual environments. Three demonstrations that have been built using EM are described, showing different design aspects of EM. Finally, we discuss several problems encountered in the development process, describe ideas for future work in order to improve EM further, and compare EM with some other networked VR systems.

Chapter 2

Existing Networked VR Systems And Tools

In this section, several design issues related to networked virtual worlds are discussed along with a case study of several networked virtual reality systems.

2.1 Centralized Or Replicated Architecture

Any architecture that distributes an interface among several users must address the choice between the two alternatives: *centralized* or *replicated* [Gre91, CMe90].

In a **centralized** architecture, only one copy of the shared application exists – in a central server (*central box*) which could be a conference agent or a dialogue manager process. The role of this central server is to multiplex shared application output to each participant's workstation, including the local screen, and to multiplex every participant's input and forward it to the shared application. In a **replicated** architecture, the central server is replicated on every participant process. As a result, each participant process has a copy of the shared application. The server for every participant accepts input from other workstations, and passes the input to its resident application, which then recomputes and generates the necessary output.

Advantages and disadvantages of these two alternatives are summarized as follows:

Centralized Architecture

- *Advantages:*

  It is intrinsically simpler. The synchronization is handled in one place which is at the central server and therefore is easy to realize.

- *Disadvantages:*

  Network traffic load is high since both input and output must be broadcast to every other workstation. Because all the activities are channeled through the central server, it will become the bottleneck as the number of participants grows. In addition, since the whole system is very sensitive to network latency and highly dependent on the central server, it could be less fail-safe than the replicated architecture.

Replicated Architecture

- *Advantages:*

  Lower network bandwidth is required because only the input, rather than both input and output, needs to be distributed among users. All participants in the environment receive a good interactive performance because they interact with local copies of the application. Local generation of output simplifies operations in heterogeneous environments. The replicated application remains single-threaded, simply multi-casting occasional announcements of its state to other users.

- *Disadvantages:*

  It is more complicated than the centralized architecture and therefore more difficult to maintain consistency across all copies of the shared application. Control mechanisms are needed to ensure that every client sees the same sequence of input events. All copies of the shared application must be synchronized and the output to users must be consistent.

Among the systems that will be discussed in section 2.3, only Rubber Rocks [CNT92], developed by IBM researchers, uses the centralized architecture.

2.2 Different And Identical Local Databases

Current multi-user virtual worlds can be divided into two groups. The systems that fall into the first group are those in which the database (content), or the set of objects of all virtual worlds, is the same. Different users may be looking at different parts of the virtual space, but they all have the same objects loaded in their local virtual world database. The systems which fall into the second group are those in which local virtual worlds are not restricted to sharing an identical set of objects. The objects loaded in the local database may differ from the objects loaded in another virtual world. All centralized multi-user virtual worlds belong to the first group, since there is only one copy of the application and data for all the users. Among the replicated systems that will be discussed in section 2.3, only BrickNet [SSPN94] falls in the second group.

Whether identical or different local databases are used is highly dependent on the application itself. For Computer-Supported Cooperative Work (CSCW), Tang derived six criteria that a shared work environment should support [Tan91]. One of those criteria is that a collaborative work environment should enable all participants to share a common view of the work surface while providing simultaneous access and a sense of close approximation to the work surface. So virtual worlds acting as cooperative work environments, such as Rubber Rock [CNT92], DIVE [CH93a], SIMNET [CDG93] and Groupware systems [EGR91], all have identical local databases. They are called *multi-user-same-content* applications.

BrickNet [SSPN94] is aimed at building a network-based design environment where a complete design task is distributed over several design nodes. Each node is responsible for a part of the design and shares its design objects facilitating collaboration on the larger design task. BrickNet doesn't force users to have an identical local database and share a common object space.

2.3 Case Studies

In this section, some of the previously developed networked virtual worlds and toolkits for constructing networked virtual reality are examined based on their architectures and principles.

2.3.1 VR-DECK

The Virtual Reality Distributed Environment and Construction Kit (VR-DECK) developed at IBM Research is, like the MR Toolkit, an integrated system for designing and constructing virtual worlds [CJKL93]. It eases the development work while supporting distributed computing, multi-user capability and a variety of I/O devices. VR-DECK consists of *modules, events and rules.* It is an advance on the architecture of the previously developed centralized system – Rubber Rock [CNT92], because it distributes the role of the central dialog manager among a collection of peer processes called modules. Modules are the building blocks (discrete objects or a collection of objects) of the virtual environment.

All communication between modules is done by exchanging (producing and consuming) events, which are defined at a higher level using rules. Each module has a set of rules which determines which event to accept, what type of event to generate, and how the module reacts to a certain event. The inter-module network traffic is minimized by intersecting the set of events that one peer is interested in with the set of events that all other peers are interested in, so that only interested modules receive events from one another. The modules communicate with each other through TCP sockets.

Rubber Rock [CNT92] is a multi-user virtual world where more than one user can manipulate several flexible objects simultaneously. The simulation in Rubber Rock is centralized (it used centralized architecture), and users connect to the central server to generate their own view of the system in action.

2.3.2 SIMNET, NPSNET And VERN

The most famous example of multi-user virtual environments is SIMNET, the SIMulation NETworking system developed by LORAL Advanced Distributed Simulations Inc. It was initiated in 1983 as a project on large-scale networking simulators by the Defense Advanced Research Projects Agency (DARPA) [All91]. The goal was to develop the technology for building a large-scale network of interactive manned platform combat simulators for team training and battlefield developments [SLA92]. SIMNET was *successfully* concluded in 1990, and currently its cites locates at more than 250 locations in the United States and Europe. The key design principles [CDG93] listed below, together with the distributed combined arms and tactical team trainer prototypes, form the basis of the current Distributed Interactive Simulation (DIS) technology applied to team training. The DIS standard has recently been accepted by the IEEE as the 1278 standard for distributed simulations [SLA92]. The key design principles that created the foundation for the SIMNET architecture are [CDG93]:

- The model is object/modular and event based.

  Similar to VR-DECK, the virtual world consists of a collection of objects that communicate with each other by exchanging events. Static objects such as trees, mountains and terrain are known by all the dynamic objects. The dynamic component of the world is updated by all active objects through events which declare changes in state.

- Objects are autonomous.

  Unlike VR-DECK where the sender knows which module might be interested in the event that is going to be sent out, in this model, an object broadcasts an event to all objects without calculating which other objects might be interested in the event, or how the receiving objects might be affected by it. It is the responsibility of the receiving object to decide what it is going to do about the received event. A

side benefit of using autonomous objects is that peers may join or leave a virtual world at any time.

- Objects transmit the absolute truth about their state.

  Receiving objects are responsible for determining whether they can perceive an event or not and whether they are affected by it. Before presenting the state information to a human or to a sensor system, the receiving object should transform "the absolute truth" information to model the real world, for example, to decide the visibility of a tank.

- Objects transmit information only about *changes* in their state

  To minimize the amount of network traffic and the requirements imposed on communication processing and bandwidth, objects transmit only *changes* in their behaviors.

- Objects are capable of extrapolating state by "dead reckoning".

  Since only changes about state are transmitted, each object extrapolates the new positions for remote objects from the old state rather than constantly receiving state updates. Sending objects are responsible for determining appropriate intervals between updates before discrepancies become too large. This extrapolation technique is known as "dead-reckoning", and is very efficient compared to sending data at fixed intervals.

SIMNET can run on Local Area Networks (LANs), Wide Area Networks (WANs) and satellite links joining hundreds of independent simulated components, such as tanks and aircraft together. SIMNET is indeed, as Howard Rheingold put it, "video games for real warriors" [How91]. Probably, its system architecture is "not an off-the-shelf solution for all the challenges inherent in future virtual realities, worlds

and cyberspaces" [CDG93], but the principles behind the architecture and SIMNET developing experience can be used to develop networked virtual environments.

NPSNET, developed by Navel Postgraduate School [ZPW92] and VERN, developed by University of Central Florida [BMC92] are based on SIMNET networking technology. One important contribution of NPSNET is its research to make networked virtual reality more affordable. VERN is SIMNET's extensible object-oriented class hierarchy where the communications, dead reckoning and process control are abstracted to the highest levels of hierarchy.

2.3.3 DIVE

The Distributed Interactive Virtual Environment(DIVE), developed at the Distributed System Laboratory, Swedish Institute of Computer Science, is a platform for heterogeneous multi-user virtual environments [CH93a]. Its developers used the concept of *a shared memory* to describe their model. A DIVE *process* represents either a human user or an application. A set of processes interact by making concurrent accesses to a shared memory and by sending messages to each other. Each DIVE object has a unique identifier and contains built-in behaviors, just like an OML object (see section 3.2). When objects interact with each other, the transmitted message may trigger some behaviors resulting in changes in the object transformation and material.

A process group in DIVE is a set of processes which can be addressed as one entity: multi-cast protocols can relay the messages addressed to the group as one. The prime advantage with this process group abstraction is that, because a single multi-cast data packet can transmit data to a number of different processes, atomic multi-cast protocols can be used so that each process can receive exactly the same updates to its image of the world with reduced network traffic. This achieves high availability of replicated data by fault-tolerant computing algorithms. Like the MR Toolkit peer package, DIVE uses *peer-to-peer* data distributions and communications between processes.

In DIVE, there are three mechanisms to ensure consistency in the replicated database: mutual exclusive locks, reliable source ordered multi-cast and distributed locks.

Several applications have been built with DIVE, such as multi-user 3D drawing tools, robot manipulators and 3D air traffic control systems. The use of active replication in DIVE has proven to be more powerful than the centralized architecture.

2.3.4 BrickNet

BrickNet is a networked virtual reality toolkit developed at the National University of Singapore [SSPN94]. BrickNet uses a replicated architecture for shared data distribution. Different from DIVE, which uses a peer-to-peer communication scheme, BrickNet has a client/server communication configuration. A virtual world developed using BrickNet is a *client*, which is connected to a *server* to request objects of interest and communicate with other clients. Thus, a BrickNet server acts as "an object broker and to communication server for its clients" [SSPN94].

As mentioned in section 2.2, BrickNet is aimed at a large scale design task consisting of many sub-tasks which are distributed over several design nodes. BrickNet virtual worlds therefore are not restricted to having identical local databases (set of objects), as is the case with SIMNET, VR-DECK, NPSNET and VERN. A virtual world (client) manages its own set of objects, some or all of which may be shared with other clients over the network. This means a client may *request (lease)* objects from a server or *donate (deposit)* objects of its own to be shared. If the required object is not on the server, the server will borrow it from other servers over the network. These transactions are hidden from the client.

Clients run asynchronously without waiting for the data to arrive from the server and execute autonomously, but all the message exchanges are via the servers. Concurrency control is handled at the server by ensuring the same operations are performed in the same order for all clients. This "client/server" communication configuration is similar to that in WAVES [Kaz93] which is a VR architecture developed by the

University of Waterloo. In WAVES the "client" is called a "host" and the "server" is called a "message manager". BrickNet uses UDP sockets to transmit messages, which is much the same as the MR Toolkit peer package.

Chapter 3

Background

This thesis is an extension of the software package developed at the Computer Graphics Laboratory, University of Alberta for the production of both single-user and networked Virtual Reality (VR) applications. It is based on existing tool components, including the MR Toolkit, the MR Toolkit peer package, and OML. They provide the base technology which allows developers to construct single-user virtual environments without having to develop the basic support software. This has already allowed a wide range of researchers to work in such diverse areas as scientific/information visualization, simulation, training, computer aided design, art and entertainment.

3.1 Minimal Reality Toolkit (MR)

The **MR Toolkit** (Minimal Reality Toolkit) is a subroutine library that supports the development of Virtual Reality (VR) user interfaces [GSW93]. It provides support for common VR devices such as the Polhemus Isotrak Tracker, Shooting Star ADL-1, the VPL DataGlove, the VPL EyePhone and Virtual Research Head Mounted Displays(HMDs). MR also supports distribution of the user interface and data over several workstations. Numerous interaction techniques, physical and virtual geometry management, and real-time performance analysis tools are also provided in MR.

The MR Toolkit simplifies the production of VR applications by providing stan-

dard facilities required by a wide range of VR user interfaces. It provides a software architecture called the *decoupled simulation model* that effectively supports VR [SGLS93] by partitioning the virtual world into four components: *presentation, interaction, geometric model and computation.* The *computation* component is the application's core, proceeding independently of the other system components and generating data for the geometric model component. The *geometric model* component maintains high-level representations of the data in the computation and converts the data into a form amenable to output by the presentation component. The *presentation* component produces the views of the application data that the user sees. The *interaction* component manages all the inputs from the user and coordinates all output to the user. The computation component runs asynchronously from the presentation component's visual and other update rates.

The MR Toolkit supports distributing the above four components across workstations on a network. It distributes the computation or simulation part of the virtual world as well as device handling. Among all the MR processes, one is the **master** process, others are **slave** or computation processes. The master process establishes socket connections with slave processes, gets information from the user and displays the output to one of the user's eyes. Slave processes are responsible for generating outputs to the other eye when *stereo* mode is used.

An MR program is divided into the configuration section and the computation section, where the former specifies the resources required by the environment and the latter runs the actual application.

3.2 Object Modeling Language

OML (Object Modeling Language) is a procedural programming language, with extensions for geometry, object-oriented programming and behavior specifications [Gre94]. It is used to describe the geometries and behaviors of three dimensional objects used in virtual worlds. The geometrical modeling aspect of OML are independent of any particular graphics package, and interfaces currently exist for the graphics languages

GL and PHIGS. The OML compiler is used to produce an interpretable version of the object specification (prototype) and the OML interpreter is linked with the application program at run-time. In this way one can create an arbitrary number of instances at run-time and have high level control over their behaviors. Different instances for the same object can have different geometries and behaviors, since object specification can be parameterized. The following code shows a simple example of an OML file for object Box_OBJ. The definition of an object consists of three parts: *parameters* (which could be empty), *geometry,* and its *behavior* (which could be empty).

```
object Box_OBJ
  parameters
      Bx : double;
      By : double;
      Bz : double;
  end
  geometry
  var
      aColor : color;
      aPolygon : polygon;
      aMaterial : material;
      aSolid : solid;
      aTransform : transform;
      box_obj : solid;
  begin
      aPolygon = box(1.000000,0.800000,1.200000);
      aColor = (0.992156869, 0.192156869, 0.192156869);
      aMaterial = newmaterial(aColor,0.2,0.5,0.3,100.0);
      settransparency(aMaterial,1.0);
      setmaterial(aPolygon, aMaterial);
      box_obj = newsolid();
```

```
        addsolid(box_obj,aPolygon);
        translate(box_obj, Bx, By, Bz);
        convert(box_obj);

    end
    behavior

        rotating
        var
            angle : double;
        begin
            angle = 0.3;
            rotx(box_obj, angle);
        end

    end

end object
```

The *parameter* part defines the parameters that are global to the entire object definition allowing the environment designer to customize its appearance when an instance of the object is created. As a result, each instance can have unique characteristics such as a distinct geometry, color or velocity. In the example OML code, the three parameters "Bx", "By" and "Bz" represent the position of the object Box_OBJ in the environment.

The *geometry* part describes the appearance of the object in the environment. It can be static or dynamic where the object's appearance changes in response to *events*. Events may be generated by either the system (e.g. clock ticks) or the application (such as a hand gesture and a collision with an instance). The conditions that generate events are re-evaluated every time step. The geometry of object Box_OBJ shown in the sample code, simply contains a primitive called "box".

The *behavior* part describes how the object responds to the events that occur in the virtual world. It consists of a procedure performing actions for the behavior. The actions can be dynamic changes to the object geometry – which involves modifying transformations between the object's sub-parts, or changing some aspects of the

object geometry. If no behavior is specified for an object, it cannot respond to any event. The parameters, variables and expressions are evaluated at run-time when an instance of the object is created. The object Box_OBJ shown in the sample code has a behavior called "rotating", in which the action is to rotate the object for a fixed angle continuously at each time step.

3.3 Development Of Single User VR Using OML And MR

A single user virtual environment contains instances of several objects. The object descriptions, written in OML, are processed by the OML compiler to produce a file of object prototypes. This file contains all the information required to generate instances of the object and interact with those instances. There are two sections for OML interpreter to construct a user program's virtual environment – initialization section and application section. During the initialization section, the OML interpreter reads the object prototype files and produces the geometry for each instance. The user program provides the OML interpreter with each instance's initialization parameters, if there are any. During the application section, the OML interpreter invokes instance behaviors and displays them on the screen.

The behaviors for the instances are implemented by collections of procedures. These procedures can be invoked at the appropriate times after the simulation is started. The OML run-time system supports the uses of an **event table** to automatically invoke behavior procedures when particular conditions or events occur. Each instance has an event table associated with it. This table has one entry for each condition that automatically invokes a behavior procedure. Each table entry has three components: an event code, a condition procedure and a behavior procedure.

A number of events can be generated during a simulation. Events can be generated by the system itself, such as clock ticks, by making particular gestures, or by collision

detection. A condition procedure is an OML procedure which has no parameters and returns an integer value. This procedure determines whether a particular condition has occured in the simulation and then returns a non-zero value if it has, or zero if it has not.

At each time step, the C program processes the events in the event table for all the instances. Once the event occurs or the condition under which the behavior should be activated becomes true, the state of the behavior is set to be active and added to the active list. Then at the next time step, all the behavior procedures on the active list are executed.

Overall, the development of a single user virtual environment using MR and OML consists of the following 10 major steps:

1. *Starting MR configuration section.*
2. *Specifying the devices used by the application.*
3. *Indicating the end of the configuration section.*
4. *Establishing the mapping from the room coordinate system to the environment coordinate system.*
5. *Calibrating the devices.*
6. *Loading in OML files containing object prototypes.*
7. *Setting parameters and creating OML instances.*
8. *Establishing event tables for instances which have behaviors.*
9. *Creating an application loop, consisting of reading all the input devices, generating the response to the user's input, starting the MR display loop, processing OML events, displaying the OML instances and terminating the MR display loop.*
10. *Terminating MR, closing all the devices.*

3.4 The MR Toolkit Peer Package

The MR Toolkit Peer Package is an extension to the MR toolkit that provides the *connection level* facilities to allow multiple independent MR applications to exchange data with one another across the Internet [Sha93]. The master process of MR, or the **peer**, can transmit device data to other remote master processes and receive device data from them. Each MR application only has one master process. Application-specific data can also be shared between peers. Any MR Toolkit program may start up the peer package at any time, and may initiate and quit communications with other processes at will. Peers are connected pairwise and one peer may send a message to any or all other peers through procedures.

A local process can update all active remote peers with the most up-to-date local device data and accept all incoming peer messages, automatically update local copies of remote device locations, sort incoming shared data and manage connections. Peer shared data is the basic tool for transmitting and receiving application-dependent data from an attached peer (this is used later in this thesis implementation).

For the following three reasons, the peer package uses the User Datagram Protocol (UDP) for network communication: (1) connection management is simple, (2) packet communication is one-way, (3) communications errors do not result in error returns from data sending calls [SG93a]. The UDP protocol is connectionless, so there is no connection establishment and termination overhead. Packet communication is one-way, there is no acknowledgment for the data sent by the peer. But, using one-way packet traffic for sending device data can reduce device-centered lag substantially [LSG91]. Indeed, one-way packet communication is quite valuable in real-time applications. Reliable TCP (Transport Control Protocol) protocol does not satisfy the real-time requirement better than UDP in the sense that it takes longer to transmit data, due to the fact that it has to wait for acknowledgments for the data. However, the UDP protocol is not "reliable", which means that it does not guarantee the receiver will get all the transmitted data, nor does it guarantee the data packets will be received in the same order that they were transmitted. So peer-based shared

data carries some risks.

Each peer is explicitly connected to all other peers, so by default the peer package maintains a complete graph connection topology. The advantages of the complete graph topology is that the peer package need not concern itself with routing, or with forwarding incoming peer packets to other peers. The clear disadvantage of this topology is that the number of packets transmitted is on the order of the square of the number of participants (known as the N-squared problem) [SG93a]. In many aspects, it would be much better to use the Internet multi-cast facility [All91, SG93a], where traffic flow grows linearly with the number of participants. However, a certain amount of system management is required to support multi-cast beyond a local-area network and this is difficult to achieve for a wide range of users without super-user privileges. For this reason, multi-casting wasn't used in the peer package. As it stands, the peer package can be re-implemented to use multi-casting without changing its user interface.

3.5 The User Interfaces

MR Toolkit and its peer package is a subroutine library. The OML interface is a set of C procedures. As a result, in order to initialize and run a virtual world using MR and OML, a considerable amount of C programming is required. The Environment Manager developed by this thesis work, enables the programmer to interact with a simple EM description file instead.

Chapter 4

Building The Environment Manager(EM)

4.1 Introduction

The object modeling language (OML) is used for the creation of 3D hierarchical objects with procedural functions and animation. When an environment is coded using OML, much of the work is spent designing geometries. A highly interactive modeling and animation CAD system – JDCAD+ has been developed in the Graphics Lab at the University of Alberta [Sea93b] to enable the user to design an object for a virtual world with appropriate details, but with much less effort. In addition to providing users with a geometry editing tool, it also provides a key frame animator to generate procedural code for the object's behavior. A translator program has been developed to convert JDCAD+ output into OML code [Sea93a]. The OML file generated by JDCAD+ contains geometry and behaviors for only one object. Usually one virtual environment contains several instances of the same or different objects. As summarized in section 3.3, in order to build a single-user virtual world, the user has to:

- integrate several OML files;

- manage to load in multiple objects and create one or several instances of an object with different parameters;
- control the instance behaviors;
- set up a VR environment using MR and OML in one or several C programs.

For a complicated system, the above list involves quite a large amount of work as the user must learn many things – such as how to use the underlying MR functions, OML and the C programming interface for creating, controlling and simulating OML objects, instances, and behaviors. A question often asked by developers is – "Is it possible to build and run a virtual environment without writing any C or MR code?". The answer is "Yes". This is exactly what the Environment Manager (EM) does for users. All the user has to do is provide an environment description file. EM provides higher level support and hides the underlying implementations. The relationship between EM, OML and MR is illustrated in Figure 4.1.

The initial idea of EM was created and implemented by Mark Green [Gre93], but the first implementation version had some errors and unexpected features. A more intelligent, flexible, accurate and robust parser was needed. The first EM version also did not support the creation of multi-user virtual worlds. This thesis work includes the design and implementation of:

- The networked virtual reality system architecture; two concurrency control schemes; interaction techniques; and communication protocols for shared information, such as variables, instances, and objects.
- A two-pass EM description file parser to parse the description file when EM starts the virtual world, and to parse the sub-EM description file code that EM receives at run-time.
- A mechanism for detecting collisions of an instance – with all the instances of a specified object, with all other objects/instances except a specified object/instance, and with any instance except itself.

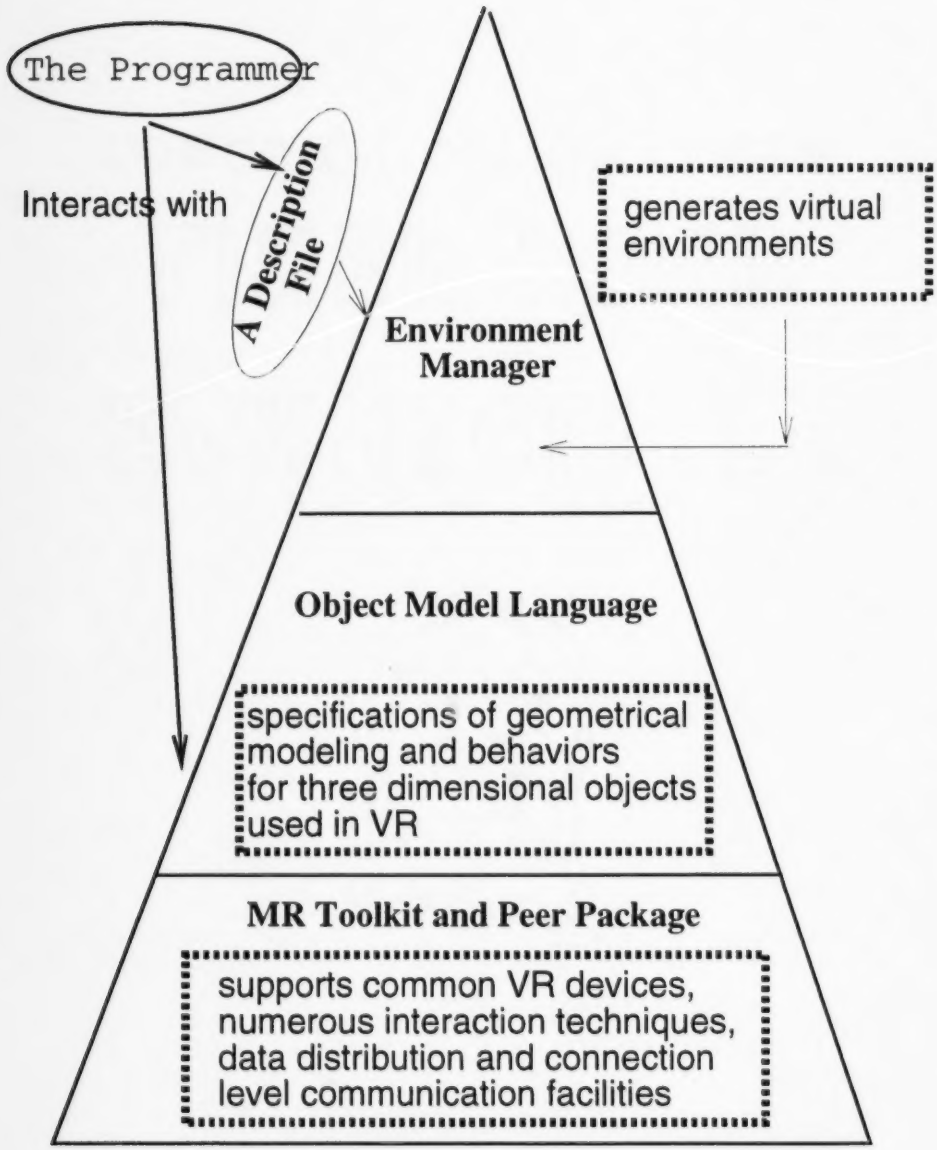

The Programmer
Interacts with
A Description File
generates virtual environments
Environment Manager
Object Model Language
specifications of geometrical modeling and behaviors for three dimensional objects used in VR
MR Toolkit and Peer Package
supports common VR devices, numerous interaction techniques, data distribution and connection level communication facilities

Figure 4.1: EM, OML, MR And Their Relationships

- Behaviors for objects to be inherited by their instances.
- Three networked VR demonstrations developed by the new environment manager to investigate the capability of EM.

In the following sections, we discuss several EM design and implementation issues: (1) Architecture and components (2) Concurrency control (3) Unique instances (4) Reduction of bandwidth (5) Shared information.

4.2 Networked VR System Architecture

In this section, the replicated application architecture, object space configuration, and the organization and components of the Environment Manager (EM) are described.

4.2.1 Replicated Application Architecture

In section 2.1, we gave definitions of the **centralized** and **replicated** architecture and compared them. Application replication offers two key advantages over the centralized architecture: **interactive performance** and **versatility**. However, it requires that all replicas of the application be synchronized at all times. One key aspect of structuring networked virtual reality is:

> To ensure that the network does not sink the application. Response time is vital to the success of networked VR

In order to have better responsiveness, EM applies a *replicated application* architecture.

In section 1.2.2, we introduced the *decoupled simulation model(DSM)* – the software architecture of the MR Toolkit. In DSM the *Computation* component receives syntactically valid user commands as input from the *Interaction* component and updates the application *asynchronously* from the *Presentation* component's visual and other update rates. The decoupling of the dependency of the visual update rate from

the application update rate results in significant system speed up. The Environment Manager sits on top of the DSM and is in charge of the initialization and management of the local application as well as communications between the other applications for information exchanges. The replicated application system architecture is shown in Figure 4.2.

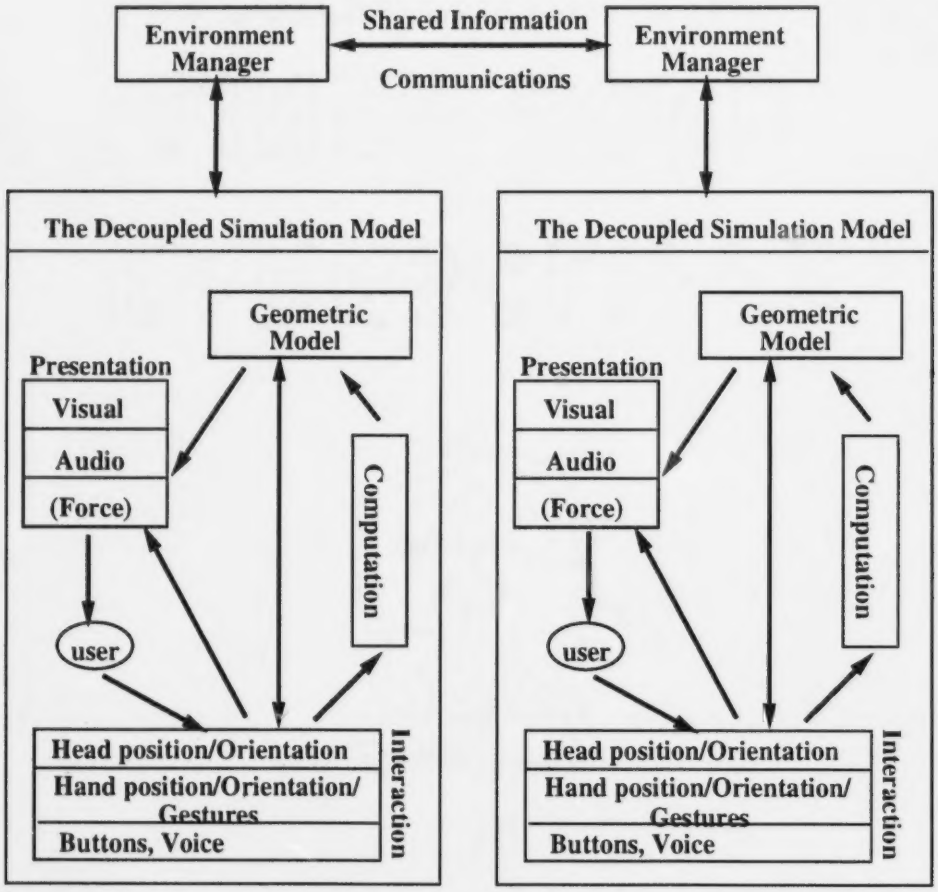


Figure 4.2: Application Replication Architecture

4.2.2 Configuration Of Object Space

The networked virtual world built by EM (called an **EM world** in the rest of this thesis) is comprised of OML objects, and instances that embody their graphical, behavioral and network properties. Being created by OML, which is an interpreted, object-based language, instances in EM worlds can be added or deleted, and their attributes can be modified at runtime. Unlike SIMNET, EM worlds are not restricted to sharing an identical set of objects. An EM world can manage its own set of objects and instances, some or all of which may be shared with other EM worlds on the network. The objects which reside on the user's home machine are called **local objects**. The instances which are owned by the machine where the application is running are called **local instances**. Those objects or instances that are shared with other applications are called **shared** objects or instances. They are the representation or *ghosts* used in the other virtual worlds for the local objects or instances. To help understand the architecture of EM worlds, the following is a brief description of two EM worlds. The details will be presented in chapter 5.

The first example is a tank battle simulation environment (a simplified SIMNET). In this demonstration each virtual world has the same object database, all the instances are shared by different worlds. Different tanks owned by different teams are represented by different colors. All the tanks are different instances of the same object – "Tank" which resides on each driver's machine (of course, we can have different tank objects to represent different types of tanks). So each driver (an EM world) will send out the representation of its own local tank (the local tank's *ghost* instance) to other drivers (other EM worlds). As a result, in each driver's view, he/she can see all the tanks which are currently in the simulation. Each driver can manipulate his/her own tank by driving it, changing its direction and, at the same time, can also aim at or shoot any other ghost instance reflecting some other driver's tank. The network configuration for this demonstration is shown in Figure 4.3.

The second demonstration is the East Edmonton Mall (the competitor with the West Edmonton Mall) Design Environment. It is a simplified example of a network-

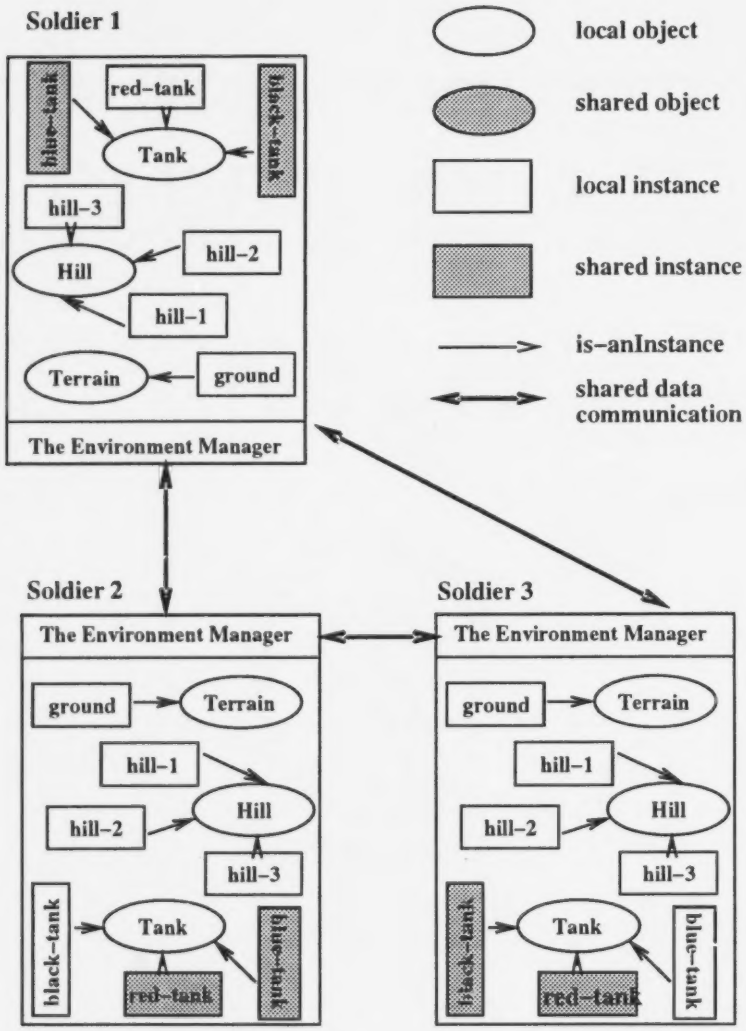


Figure 4.3: Networked Configuration Of The Tank Simulation

based design environment where a complete design task is distributed over several design nodes. As illustrated in Figure 4.4, the whole example design system is comprised of two design studios and a plane museum (each of which is represented by an EM virtual world) at different geographical sites. Each studio or museum has its own *master pieces* (e.g. biplane model and statue "smiling") that can be **deposited** with the other studios. These objects are *local objects*. Each studio can also **lease** master pieces deposited by the other studios or museums, in order to integrate the pieces into a complete design work. These leased master pieces are *shared objects*. In this demonstration, each EM world has a different content and object space. OML code for a shared object will be transferred over the network in order to both create and update the shared object in another world. The network demonstration's configuration is shown in Figure 4.4.

4.2.3 Architecture And Components Of The Environment Manager

The Environment Manager (EM) consists of four subparts: the description file parser, interaction support, the VR world and the communication handler (see Figure 4.5).

The Description File Parser

This subpart contains a two-pass EM parser (implemented using YACC) for parsing EM world description files. The parser reports syntax errors in the description file and extracts the information about local OML object files, device set up, choice of concurrency control scheme, leased OML objects and instances, OML object behaviors, OML instance parameters, behaviors, and shared variables. All this information is directed to other corresponding EM subparts to process.

The description file can be divided into the following six parts (see Appendix A.2):

1. EM world device configuration.

   In this part, the user specifies the devices that are going to be used and which

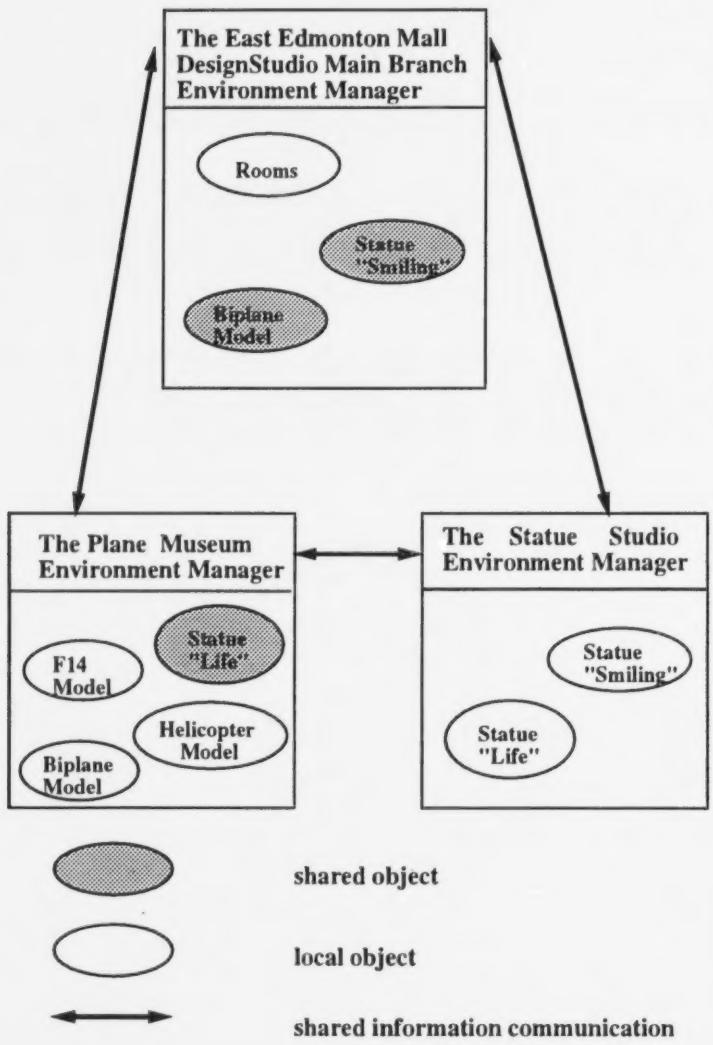


Figure 4.4: Networked Configuration Of The Design Studios

device will be used to track the hand position and orientation. EM parses the device information and starts up the appropriate device set using the MR toolkit functions.

2. Networked EM world description.

   In this part, the user specifies:

   - the concurrency control scheme the EM world will use.

     EM makes several concurrency schemes available for the user to choose from. *Simulation ownership token parsing*, also known as *round-robin*, is one of them.

   - shared object information.

     It includes a list of the names of the objects that are required by this EM world. Hopefully they can be obtained from other EM worlds which have them and are willing to *deposit* them.

   - shared instance information.

     This information should be provided when an EM world wants to send shared instances to other EM worlds. Of course, those EM worlds should have the object database that the instances are created from. The information is a subpart of the EM description file containing instance entries, each of which includes: the object name, instance name, parameter values, behaviors, and shared variables.

   - expected instance information.

     It includes a list of the names of the objects, other than the required objects, that an EM world is interested in. The previous item includes the instances sent out by the EM world. Instances that the EM world is interested in receiving are specified here, in terms of their object names.

     By utilizing this information, together with the information about the required objects, an EM world can decide whether a received instance

should be accepted or discarded.

3. OML object files list.

   In this list, the user tells EM where to find all the OML object prototype files which will be the object database for the EM world to be constructed. EM loads in these OML object prototypes, generates the object database and later, the instances which form the content of the EM world are created from the database.

4. OML object behaviors list.

   Object behaviors, if there are any, are defined in this list. The reasons for having object behaviors are:

   (a) If several instances of the same object have the same behavior, there is no point in writing redundant lines in the instance list. Instead, we can define this information as *objects behaviors* in its object. Every instance can inherit these behaviors from its object if the instance doesn't specify its own behaviors.

   (b) For multi-user-same-content applications (defined in section 2.2), when an EM world receives a shared instance, and if its behaviors are the same as the other instances of the same object, the object behaviors that have already been defined in the receiving EM world, can be executed as the shared instance's behaviors. Therefore, the sending EM world doesn't need to send out the behaviors for the shared instance and network bandwidth can be saved.

   (c) When an EM world receives a shared object, it is usually the sending world's responsibility to transmit the behaviors of the object, because the sender is the owner of the object. Every subsequent instance of the shared object that the receiving EM world creates, will inherit these behaviors.

5. OML instance list.

This list contains an entry for each instance created at run-time. Each entry consists of the name of the object, the name of the instance, the parameter values used when the instance is created, the mapping from events and conditions to behaviors, and the shared variables for the instance, if there are any. Here is an example of the instance tank1 entry for object Tank_OBJ.

```
instance Tank_OBJ tank1 (
    Tx = 10.0,
    Ty = 10.0,
    Speed = 0.5,
    Color = (1.0, 0.0, 0.0),
    Owner_ship = 1)
actions
    tick            signal_marching
    begin_marching      marching
    tick            turning_with_hand
    tick            signal_stop
    collision ( hill_OBJ ) explosion
    collision ( Bomb_OBJ ) got_it
    collision ( tank_OBJ ) explosion
    need_to_explode explosion
end
shared information
    tx      double      readable
    ty      double      readable
    hit     integer     readable writable
    heading     double      readable
end
```

From the above information, the instance tank1 of object Tank_OBJ is created at run-time with a start position of (10.0, 10.0, z-value is fixed in this demonstration), traveling speed at 0.5, red color, and is owned by this EM world. Among its behaviors, three are triggered by tick events, three by collision detection events and two by conditions. The instance's shared variables include "tx", "ty", and "heading" which are values of OML data type **double** representing the position and the orientation of instance tank1. Their access permissions are "readable". The shared variable "hit" indicates whether the tank has been hit by other tanks and is defined as OML data type **integer**. The access permissions of variable "hit" are "readable" and "writable".

The EM description file parser has two passes. The reasons for this are:

1. To get information on the devices used before the MR configuration function is called.

   The initial implementation of the EM parser was a C language procedure. MR was started using the DataGlove and EyePhone device regardless of whether they were actually used in the simulation. Since currently the DataGlove can't be shared by two clients, the EM initial implementation caused applications to be hanged if the DataGlove has been added to the device set by another EM application, even though that application didn't actually use the DataGlove.

   Before an EM world sets up a connection to another EM world, it has to know the devices it is using. EM obtains the knowledge of the devices used in the first pass, then starts up the MR configuration section aware of the devices needed for the application. In the second pass, EM parses the objects, instances, and shared data information, then sets up peer to peer connections, creates instances, and simulates the virtual world.

2. To allow forward reference.

   Look at the following EM description code containing two entries for instance "hill1" of object "hill_OBJ" and instance tank for object "tank_OBJ".

```
%1          instance tank_OBJ tank
%2          actions
    %3          collision ( hill1 ) explosion
%4          end
%5 instance hill_OBJ hill1
```

If we use a one pass parser, by the time the parser reaches line %3: collision with hill1, where name "hill1" is a *forward reference* to an instance named "hill1" declared at line %5, there is no way for the parser to figure out whether "hill1" refers to an instance, or an object, or if it is a wrong name – because the name "hill1" hasn't been seen yet. A one pass EM parser has difficulty dealing with *forward references*, which occur quite frequently in EM description files. By adding a second pass to the parser, this problem becomes easy to solve.

Interaction Support

The MR Toolkit manages devices such as 3D position and orientation trackers, hand digitizers, and sound I/O using the client-server model. The client-server structure used for device-level software has many benefits in terms of device independence and application portability. The structure also provides support for optimal predictive and noise-reduction filters. Efficient data communications and application distribution are also achieved. How can users take advantage of all these functionalities which are all transparent to the users, while they are writing OML code? The EM interaction support provides an OML interface with the MR device client procedures for users. It creates an instance called "test_body" of an OML object called "body", where the eye position and orientation, the hand position and orientation, state information of the three buttons on the IsoTrack and the three buttons on the DataGlove, are represented by instance variables. They are:

- position

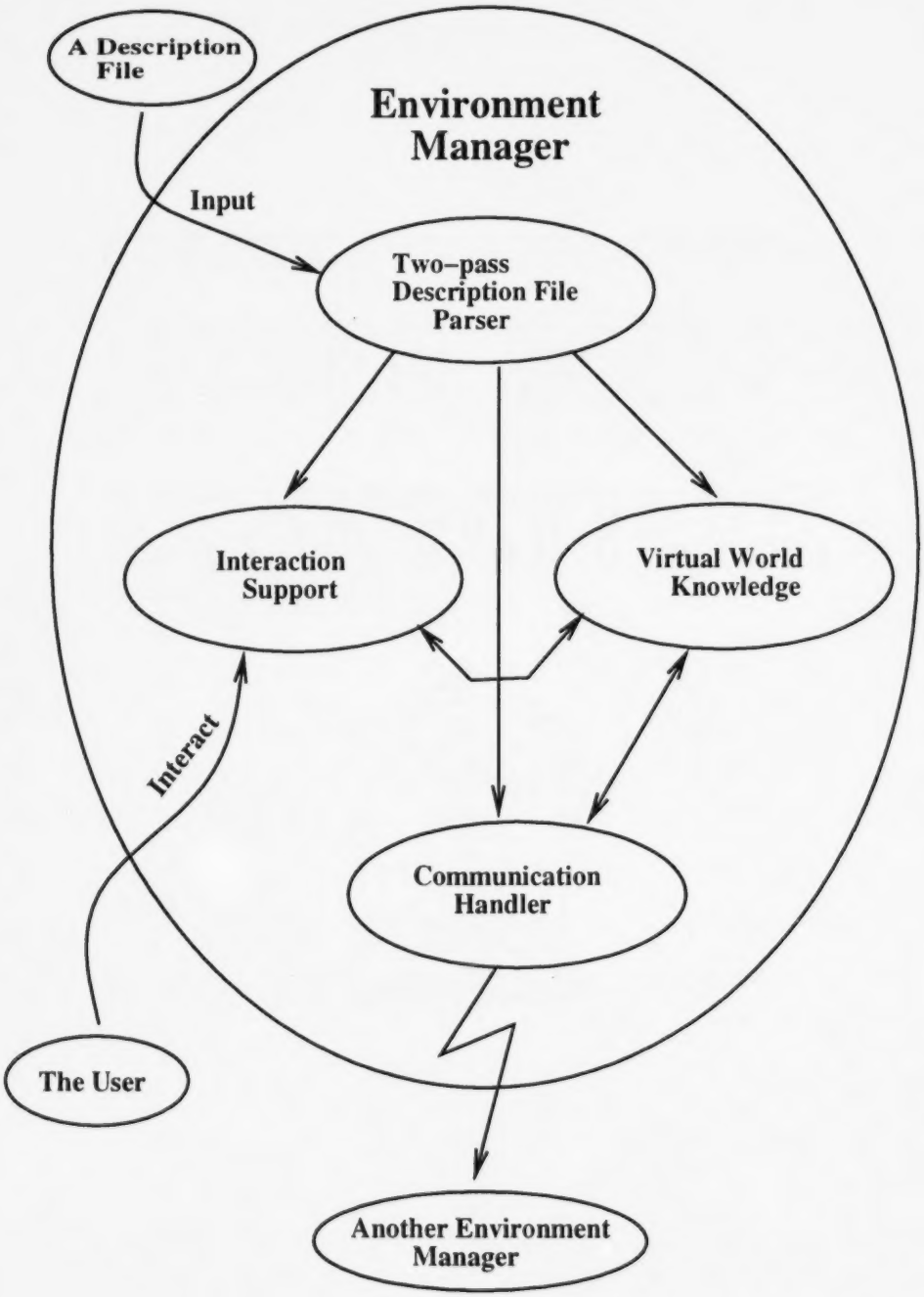


Figure 4.5: Architecture Of The Environment Manager

This is user's current eye position. It is a value of OML data type **point3** containing the three values of x-axis, y-axis, and z-axis for the eye position.

- heading

  This is the current viewing direction of the user. It is a value of OML data type **point3**. The first coordinate is a rotation about the x-axis, the second coordinate is a rotation about the y-axis, and the third coordinate is a rotation about the z-axis.

- hand_position

  This is the position of the user's hand. It is a value of OML data type **point3** and is interpreted in the same way as the "position" variable.

- hand_orientation

  This is the orientation of the user's hand. It is a value of OML data type **point3** and is interpreted in the same way as the "heading" variable.

- iso_button0, iso_button1, iso_button2

  These three variables are values of OML date type **integer**, representing the state of the three buttons on the IsoTrack. Each value normally stays as 0, unless its corresponding button is pressed, the value will be set to 1.

- glv_button0, glv_button1, glv_button2

  These three variables are values of OML date type **integer**, representing the state of the three buttons on the DataGlove. They are interpreted in the same way as the three values for IsoTrack buttons.

With regard to the hand position and orientation, their values don't necessarily have to be obtained from the DataGlove; the user can specify which tracker is used to represent the hand in the description file. At the start of each time step, each of the above variables is updated with the current value from the corresponding input device. When an application needs service from a particular device, it can simply read

the value of the corresponding variable in instance "test_body" of the "body" object. The values of the variables are retrieved and used as the values of the corresponding devices. This interaction technique can be used to modify OML object behaviors based on the user's behavior, and navigate the user through the environment, based on his/her actions or the actions of other objects in the environment. On the other hand, the user can change these values so that he/she can set the application-specific view for the environment and the application-specific way of navigating in the VR world.

The VR World

Every EM world consists of instances created from objects. This subpart of EM manages objects and instances, either *local* or *shared*, to form the content of the virtual world. Objects and instances can have behaviors. We need to develop specification techniques which enable objects' reactions to interact with each other in controlled ways. The approach that EM and OML take is called the **relation approach** [SG93b], which is based on directly modeling the interaction between instances. In this control scheme, the primitive motion unit is called a *relation*, where each relation describes one aspect of an instance's behavior. A relation is triggered by events or conditions that occur in the environment. When a relation is triggered, it generates the response. The entries for each instance's relations are specified in the description file. Each entry consists of the event or the condition that triggers the response, and the name of an OML behavior procedure that performs the response.

EM loads in all the object prototypes, creates instances with different parameter values from the prototypes, positions and orients the instances within the environment, and builds the event table for all the instances containing the relations for controlling behaviors. The mapping from events and conditions in the environment to behaviors is handled at run-time. At each time step, EM checks the events in the event table to see if any behavior needs to be activated. If an event has occurred, EM activates the corresponding behavior, puts it on the active behavior list, and runs it

at the next time step.

A virtual world can receive shared OML objects represented as OML code and shared instances represented as EM description file code during the simulation. The EM parser can be activated to parse the EM description file code whenever an EM world receives them. EM can then create shared instances and control their behaviors at run-time. The shared object and updates are sent over the network in the form of OML code that the receiving EM world loads and executes. Due to the interpreted nature of OML, an object can be loaded multiple times during the simulation, which allows incremental development or updates.

The Communication Handler

The MR Toolkit peer package [SG93a] enables one MR application to receive messages from, and send messages to other MR applications. The peer shared data facility is the basic tool for transmitting and receiving application-specific data from an attached peer. Several issues about how EM implements the communication between different applications are discussed below:

1. Peer to Peer Connection.

   In BrickNet, the communication is based on client/server architecture, whereas in EM the communication is peer to peer based, we put the "server box" in the centralized architecture down to each peer.

   From the command line, EM gets the information about the remote peer that the local application wants to connect. To start the peer package, EM calls the peer initialization routine, which declares the process's availability on the given udp port number using the given application *name*. To start communications with a remote peer, EM sends a connection solicitation message to the remote peer, then the remote process sends back a solicitation message. Receipt of a solicitation message from a remote peer makes the local state of the peer *active*, allowing messages other than solicitation to be sent to the remote peer. If a

solicitation message has not arrived, a process will keep sending the solicitation message at longer and longer intervals until the remote peer responds.

The important feature is that the local process does not block while the above repetition procedure happens. All connection solicitation and other peer traffic takes place asynchronously, so any MR application may start up the peer package at any time.

2. Shared Data Management.

   EM parses shared data items from the description file, and creates two shared data records for each item. One record points to the storage area of the data to be automatically read when a shared data item is sent to remote peers, the other record points to the storage area which is updated automatically when a shared data message arrives. For some applications, such as multiplayer handball game [SG93a], these two areas can be the same. Because EM supports general applications, we allocate two different storage areas for incoming and out-going shared data.

   The shared data storage area is different from the actual OML data storage area, so the incoming shared data doesn't overwrite the actual OML data directly. EM handles OML data updates based on the concurrency control scheme chosen by the application. The peer shared facility also allows a callback to be defined whenever a shared data item arrives from a remote peer. The callbacks are used by EM to update the actual OML variables and to activate the EM description file parser if a piece of EM description file code is received.

3. Connection Topology.

   The peer package and EM maintain a complete graph connection topology by default, which means that each peer is connected to all other peers explicitly. This is implemented by having the local peer inform all active remote peers that it has received a new remote connection. When the remote peers get this message, each of them solicits a connection from the new peer and creates the

shared data records for the new peer. This is defined by EM in the callback that is automatically called when a solicitation message is received. Every peer has a list containing all the connected peers.

Any EM world may start up the peer package at any time and may initiate and quit communications with other EM worlds at will. A *quit* command is sent out when an EM world wants to quit, causing the recipient to delete this remote peer from its connected-peer list. Each EM world also maintains a list of shared instances and objects. EM defines the callback for the *quit* message. This callback deletes the shared instances and objects owned by the remote peer sending out the *quit* message.

As mentioned in section 3.4, peers are connected pairwise. Each peer has to know which peer is the receiver of the shared data item. Currently the peer packages provide procedures for a peer to send a message to any other peer at once. The existing complete graph unicast does have some problems, such as the number of packets transmitted being on the order of the square of the number of participants, but the current peer package structure is ready to use multi-cast packet traffic without changing its interface.

4. Maintaining Interactive Response

In BrickNet [SSPN94], when the application has greater need for computing resources, it is able to defer the processing of network messages. EM receives the messages meant for the application from the network and stores them, so that the application can retrieve and process them at will. This scheme satisfies the need for maintaining interactive response in networked virtual worlds by separating graphical update from network messages. It allows the EM worlds to defer processing network messages and devote all their resources to supporting interaction when necessary. For example, normally an EM world checks for network messages once within each simulation cycle, but it can decide to check for them every other cycle, or according to any other scheme. When an EM

world does not checks for network messages once within each simulation cycle, the perceived network latency would be larger, but it would help to solve the real-time graphical update problem.

The peer package and EM allow the remote peers to update the local peer's data asynchronously and without any effect on the local user's ability. This capability has proved extremely useful in maintaining responsiveness of networked virtual worlds [SSPN94].

4.3 Concurrency Control

The replicated architecture needs concurrency control to resolve conflicts between participants' simultaneous operations. For example, conflicts happen when there is more than one participant trying to change the same instance. Approaches to concurrency control have been discussed in distributed database systems. Some of the approaches, such as explicit locking and transaction processing, are not appropriate for networked virtual environment under certain circumstances [EGR91]. Networked VR systems and distributed systems are similar in that they are distributed over a network and they are shared by multiple users. However, despite these similarities, networked VR systems have their own unique characteristics. Some design issues therefore, are different from those in distributed database systems.

4.3.1 Issues And Other Approaches

The following are some of the concurrency-related issues facing networked VR designers:

- **Responsiveness**

  For real-time and highly interactive virtual reality, good responsiveness is vital. Network lag should not be allowed to hinder the user, force the user to slow down or stop the simulation. To ensure this, two properties are required: a

short *response time* and a short *notification time*. The response time is the time it takes for a user's own environment to reflect his or her actions; the notification time is the time required for these actions to be propagated to everyone's environment.

- **Wide-Area Distribution**

  One of the main benefits of networked VR systems is that they allow people to work together in real-time, even though they are separated by great physical distances. The systems may be geographically distributed. With current communication technology, transmission time and rate for wide-area networks are significantly worse than those found in local area networks. The possible impact on response time must therefore be considered. In addition, communication failures point out the need for elastic concurrency control algorithms.

- **Data Replication**

  Because the response time demand of networked VR systems is very high, the data are usually replicated for each participant. Many potentially expensive operations such as viewing a shared object from different viewpoints, can be performed locally without leading to potentially catastrophic degradation in response time.

- **Robustness**

  Usually robustness refers to recovery from unusual circumstances. These typically are component failures, such as the failure of a site or a communication link. For networked VR systems, there is also a second form of robustness these systems must achieve: robustness to unpredictable user actions. For example, adding a new user to a set of users issuing transactions is not normally problematic, but adding a participant to a networked virtual reality system can result in a major system reconfiguration. The system's concurrency algorithm must adapt to such a reconfiguration, recovering easily from such unexpected user actions as abrupt session entries or departures.

The following discusses some of the concurrency control approaches used in distributed databases.

- **Locking**

  The solution to concurrency in this scheme is simply to lock data before it is modified. The overhead of requesting and obtaining the lock. including wait time if the data is already locked, causes a degradation in response time. The major degradation occurs when it takes a long time for the application to update the data. More general, **transaction** mechanisms solve the problem by making the changes during a transaction invisible to other users until the transaction commits. Transactions implemented by locks lead to the problems described above. Generally, long transactions are not well-suited for interactive purposes.

  These problems point out a basic philosophical difference between distributed databases and networked VR systems. The former strive to give each user the illusion of being the system's only user, while the latter strive to make each user's actions visible to others. Shielding a user from seeing the intermediate states of another's transaction is in direct opposition to the goals of networked VR applications, such as Computer-Supported Cooperative Work and Groupware systems.

- **Centralized Controller**

  Centralized Controller implements the concurrency control by introducing a centralized controller process [SSPN94] assuming that data is replicated over all users/workstations. The controller receives user requests for operations and broadcasts these requests to all users. Because the same operations are performed in the same order for all users, all copies of the data remain the same. Along with the usual problems associated with centralized components (e.g. a single point of failure and a bottleneck), several other problems also arise. Since operations are performed when they come back from the controller

rather than at the time they are requested, responsiveness is lost, and the VR world of a user issuing a request is locked until the request has been processed. Without locking, a subsequent request referring to a particular data state might be performed when the data is in a different state.

- **Dependency Detection**

  Dependency detection control [SFB87] is based on annotating data with a stamp describing the author and time of the change. Every request to change data broadcasts several things: *the new data, its stamp, and the stamp of the previous version of the data on the originating machine*. When a machine receives a message requesting a change, it first checks whether the previous stamp in the request is the same as the stamp in its database. If they are different, a "dependency conflict" is signaled and is resolved by manual interaction. The major advantage of this method is that no synchronization is necessary. Non-conflicting operations are performed immediately upon receipt, so the response time is very short. However, any method which requires user intervention to assure data integrity is vulnerable to "user errors". Also, managing concurrency control by sending out more information in each packet increases the network bandwidth overhead.

4.3.2 Concurrency Control In EM

From our observations, we found that different applications may need different concurrency control schemes. The concurrency control scheme that is best for one application, may not necessarily suit other applications very well. It is not necessary to find a generic control scheme for every type of application; instead, we can implement several schemes, and leave the choice of scheme to the users.

In EM, concurrency control is application dependent. We provided the following schemes:

Simulation Ownership Token Passing

Some networked VR systems are intended for situations where only one participant at a time owns the simulation and is active, while the other participants watch the simulation, waiting for their turns. We maintain consistency of a distributed virtual world database by restricting manipulation in such a way that only one site can perform operations to alter the status of the virtual world. This simulation ownership token passing scheme is also called *round-robin simulation.*

The site having the *simulation ownership token* is allowed to manipulate objects in the world. In EM, token passing is implemented in the same way as the multiplayer handball game developed at the Graphics Lab, University of Alberta [SG93a]. Every peer has a list of other peers sorted by process id, so that all peers use the same method to decide and agree on who is the next peer to receive the simulation and to be active. After the active peer finishes updating the data, it passes the simulation ownership token to the next peer on the connected-peer list.

When the simulation ownership message is lost, simulation deadlock will arise due to the fact that both the old owner and the new owner are waiting for simulation results, and each thinks that the other is doing the simulation. This is remedied by requiring an acknowledgment from the new owner, so that when the old owner passes the simulation ownership token to the new owner, it also sets a flag indicating that acknowledgment is expected. If it doesn't receive the acknowledgment from the new owner in a certain amount of time (time-out), it re-sends the simulation ownership token until an acknowledgment arrives. All this work is done in two token callbacks. One is called when the simulation token arrives; the other is called when the acknowledgment arrives.

To enhance system robustness when system reconfigurations such as abrupt session entries or departures occur, two callback procedures are provided. They are called on each peer whenever a new peer connects, or an old peer quits, so that the peer can be correctly added to, or deleted from, the sorted peer list.

The round-robin scheme is also one way of dealing with communication lag. Cen-

tralized simulation does not perform well when lag is significant (100 milliseconds or more), because the central site's simulation will be based on data that is N seconds old and will not match what the user sees. The traditional way of dealing with lag is by slowing the user's movements down, or by adopting a move-and-wait strategy. Neither of these approaches works well because it is the *simulation* that is behind, not the user's hand and eye position. As a result, the user would need to slow down or even stop his/her natural motions. Using *round-robin*, the simulation is structured in such a way that can it avoid or eliminate the lag problem by allowing only one user to update the simulation. This user is the simulation token holder; only he/she can locally simulate the simulation and broadcast the output. The other users have to wait for the token to arrive – in other words, wait for their turn to begin the actions. After the simulation owner finishes one simulation round, he/she sends the token to the next peer on the list, stops broadcasting, and waits for the acknowledgment for the token. This is when the communication lag becomes obvious.

Communication lag is obvious only during the simulation ownership transfer because of the network latency. When the new owner sends the acknowledgment to the old owner, the old owner sees a speed up in the simulation. The time lag between when the simulation begins to slow and when it regains full speed, is equal to the total round-trip packet delay between the old and new owners. All other peers except the new owner also see this slowed motion, because the old owner stops broadcasting simulation output after he sends the token. Once the new owner starts simulation, the motion resumes. The new owner does not see the slowdown because as soon as he/she gets the simulation ownership message, it starts simulating. As a result, the only user that will be adversely affected by lag is automatically prevented from seeing lag.

Instance Or Variable Ownership And Access Permissions

Instance or variable ownership is a characteristic of an instance or an instance variable indicating that only one EM world is permitted to have control over that value.

The ownership may be fixed at inception and won't be transferred between EM worlds during the simulation. For example, the ownership of a tank in the tank demonstration is never transferred. The ownership may shift between EM worlds however, if an application requires, e.g. the multi-player handball game.

As mentioned in section 4.2.2, an EM world may have local objects and instances which have no relation to other EM worlds, or it may deposit objects or instances to the network and share them with other EM worlds. Using ownership, we can restrict sharing in various ways. This can be done by assigning different kinds of access permissions to shared instances, a process similar to the file update management in file systems. For example, an instance might only be visible to other EM worlds, but not updatable by them; an instance may only be updated by its current owner. Updates may or may not be sent to other EM worlds which have leased out the object from the network.

EM assigns access permissions to each instance shared variable. Two kinds of permissions can be defined for a shared variable:

- Writable Permission

  With this permission, the attribute is writable and can be updated by other interested EM worlds. In other words, other interested EM worlds have write permission to this shared variable.

- Readable Permission

  With this permission, the attribute is readable and can be sent out to other interested EM worlds if required. That is to say, other interested EM worlds have read permission to this shared variable.

Every EM world has write permission to its local shared variables. *Writable* permission allows a shared variable to be updated by other interested EM worlds and its own world. *Readable* permission allows the update of a shared variable to be sent out to other EM worlds.

The next scenario shows how ownership and readable/writable permissions restrict shared information access. EM world A has deposited an instance EO that has been leased out to EM world B. EM world A owns EO and every shared variable of EO in world A has only *readable* permission (EM world A has exclusive update privilege on EO), while in EM world B, every shared variable of the corresponding leased instance has only *writable* permission. As a result, EM world B can download instance EO to look at, but it cannot modify the original instance EO in EM world A. The leased instance EO in EM world B can be updated by EM world A.

The difference between *instance/variable ownership* and *shared variable permission* is the following: for shared variable permission, once the permission has been assigned to a shared variable, it never changes during the simulation; whereas for ownership, it can shift among the instances on the fly. The instance or variable ownership restrictions take effect when there are several users with write permissions trying to change the same shared variable which is writable. (see section 6.3.2).

Some of the instance attributes are not concerned with the conflict between participants' simultaneous operations. For example, in the tank battle demonstration (see Appendix C), the "hit" flag indicates whether the tank is destroyed. Every EM world can write to this flag. From the perspective of the goal of training soldiers, it is unreasonable to prevent a tank from being destroyed when the soldier sees himself blow up the tank in his environment, even though at the same time the tank may have been blasted in another EM world. Network lag and simultaneous operation conflicts should not matter in this case. As long as a tank has not been removed from the environment, if it seems to be hit, it should be treated as being destroyed, no matter how many times it has been hit.

4.4 Unique Instances

One of the main features of EM networked worlds is the ability to create multiple new instances while the simulation is running. It should be possible to communicate between instances regardless of the locations of the sender and the receiver. Each peer

must be able to determine the instances associated with shared messages. Therefore there must be a way of assigning a unique identifier to an instance which can then be used to communicate with it. EM has the responsibility of providing this name-serving function. There are two approaches to achieving this.

The first approach is accomplished in two steps. First, the local simulation node generates a unique name based on the current time and passes it to the remote simulation. If the name is unique on the remote simulator as well, the instance is formally created; otherwise, the process repeats itself until the remote and local nodes agree upon a suitable name.

The second approach is to assign a unique identifier based on the instance name received, and information about its sender – the remote node.

EM takes the second approach. OML simulation is based on an instance table where the key is the instance name. It is quite possible that the shared instance names sent by other EM worlds conflict with each other, or with the local ones, because every simulation runs asynchronously. Upon receipt of a shared instance, EM appends its sender's host id and process id to its name and thereby generates the local unique identifier (instance name) for that instance. In this way, there is no negotiation needed to allocate a unique identifier for a shared instance. This is transparent to the sender; it is the receiver's responsibility to assure that the instance name is unique in its own environment.

4.5 Reducing Bandwidth

The main limitation on maximum performance for distributed VR systems is the bandwidth of the connections between processors in the system. It is necessary to reduce the communication between processors as much as possible. In centralized architectures, the central processor can become a bottleneck where the bandwidth and time spent processing the communications become a problem. The use of application replication architecture can solve the bottleneck problem. However, there is still a need to decrease network bandwidth. Unnecessary communications can be

eliminated by doing extra local computation to avoid fetching remote data. EM implements the following ways to reduce network traffic:

- Instances transmit only shared variables whose states have been **changed**.

  If no other specific requirement is made, EM compares the OML shared data values with their previous states and chooses the candidates to be sent out from those whose states have been changed.

- Messages are sent only to relevant EM worlds.

  EM sends messages only to those peers who have the affected object and have expressed interest in receiving updates. This strategy is very different from broadcasting messages to all participants whether the messages are relevant or not. In the broadcasting case, every VR world receives the messages, processes them, and discards the messages that it does not need. This approach leads to a significant amount of redundant network traffic and wasted processing time if every world isn't interested in the message. In our approach, the messages are filtered and sent only to relevant EM worlds, minimizing the amount of network traffic, and all messages received by an EM world are useful. This is implemented in EM by maintaining a list of relevant EM worlds (peers) for every shared variable.

- Local simulation of the behavior of some shared instances in the environment is allowed.

  A moving instance can either send updates containing the new position at regular intervals, or it can inform any interested parties that it are moving and provide the necessary information for remote parties to simulate its motion locally. SIMNET [CDG93, All91] uses a "dead reckoning" algorithm to extrapolate states. Each instance extrapolates the new positions of remote instances from the states last reported by those instances. Sending instances are responsible for generating updates before discrepancies become too large. EM supports

local extrapolation using a "dead reckoning" algorithm. Hence an EM world sends messages only when they are needed to maintain simulation accuracy.

- Provides *quenching* and *unquenching* messages

  A quenching message is one that is sent by an EM world to inform other EM worlds to stop sending certain messages for a given period of time. One situation where quenching messages can be used is when an EM world decides to stop collaborative work for a period of time and does not wish to be informed of any updates for all leased objects during the break. The EM world sends a quenching message to request that other EM worlds stop sending update messages for the designated period of time. If the period of time is not specified, then the other EM worlds wait until they receive an unquenching message from the originating EM world before resuming update messages.

4.6 Shared Information

Users can specify shared information in the EM description file (see section 4.2.3). EM enables the following information to be shared among different EM worlds over the network: (1) instance variables (2) instances (3) objects (4) object behaviors.

4.6.1 Shared Instance Variables

OML instance behaviors are implemented by changing the instance variables in various ways. OML instance variables are the basic shared elements. EM supports variables which are of type Integer, Double, Color, Point2, and Point3 to be transmitted over the network. In each *shared instance*, users can specify the OML instance variables that need to be shared. At the same time, users can also assign permissions to each of them and tell EM whether or not the shared variable needs to be extrapolated using a dead_reckoning algorithm. The EM description file parser extracts this information and directs it to the EM communication handler.

The EM communication handler creates and maintains a relevant-peer list for each of the defined shared variables. Whenever a new peer joins in, or an old peer quits, EM updates the list. As mentioned in section 4.2.3, the actual OML data storage area, and the incoming and out-going shared variable storage area are separated, so each shared data item for the shared variable is self-contained. The item includes the variable's instance name, OML data type, value, permissions, and other control information. The EM communication handler transmits shared variables and updates the actual OML data values for the instance according to the shared variable's permissions and the concurrency control scheme chosen by the user.

4.6.2 Shared Instances

For applications such as SIMNET, which have identical object databases across all the participants' sites. each participant needs to broadcast its own instance to other VR worlds. EM supports the transmission of instance definitions (including behaviors and shared variables specifications) in terms of a subpart EM description file.

At first, the EM parser on the sender side parses the information, checks for grammatical mistakes, stores the information, and broadcasts it to the network after the peer connection has been established. Once the EM communication handler on the other EM world receives the shared instances during the simulation, the EM parser is activated to parse the information, generate the shared instance and new shared data item, add the instances to the instance list of its world, and generate new entries for the shared instances in the event table. At the next time step, the shared instances will be displayed in the environment and their behaviors will be invoked.

4.6.3 Shared Objects

For applications which have different object spaces over different EM worlds, EM enables OML object code to be shared over the network. Due to the fact that the MR peer package is unreliable, packets can be lost along the way, or can arrive at the

destination out of order. Reliability is guaranteed by the sharing procedure which is divided into seven steps as follows:

1. Requesting Objects

   Each EM world can specify the required objects in the EM description file. The EM parser gets this information, stores it and broadcasts it out to the network after the peer connection has been established.

   In the EM description file, users can define instances of shared objects. The EM parser decides whether an instance is of the shared objects or not, and if it is, the EM parser checks for grammatical mistakes, and stores information so that the instance can be created later when the actual object arrives (see section 4.6.4).

2. Processing Object Request

   When an EM world receives the object request, it searches its own object database for every object in the request list, then adds all the requested objects which are in its object space, and the requesting peer to a ready list called the "deposited-objects-list".

3. Checking Object Ready List and Sending Out Header Information

   The OML object code for an object is usually very big (over 100k bytes), and it is not practical to transfer the object code in one packet. Instead, the code has to be sent out in several packets. EM guarantees OML object code transmission by designing and implementing a reliable protocol dealing with packet loss and out of order delivering.

   In order to keep a balance of packet sending and consuming speed, EM only allows one packet of one object to be sent out at each time step for each peer. The next object won't be transmitted until the previous object in the ready list gets its acknowledgment.

Once an object is ready, EM sends out the header containing the object name and the size of the object code to the receiver first, then waits for the acknowledgment.

4. Getting Ready to Receive the Object

   Once an EM world receives the object header information, it allocates memory for the incoming object according to the size given in the header. This memory together with the object name and a flag array used to indicate which packet has arrived, are added to a list called the "receiving-objects-list". After this, the receiver replies to the sender with an acknowledgment for the header information and is ready to accept the object code.

   If there is more than one EM world having the same requested object, the requesting world takes the first one it receives and ignores those that arrive after. When EM discovers an object header with a required object name that has already been processed, it informs the sender that the object has been obtained. As a result, the sender won't deposit the object again and the receiver always has one lease for each required object.

5. Transmitting OML Code Packet by Packet

   On the object sender's side, once it gets the acknowledgment for the object header information, EM transmits the OML code by dividing it into several packets and transferring only one packet at a time to the peer that made the request. EM puts a sequence number and the object name in each packet.

6. Processing Arrived Object Code

   On the object receiver's side, once it gets an OML object packet, EM puts the packet in the appropriate memory according to the object name and sequence number given in the packet. The flag array element (mentioned in step 4) indexed by the packet sequence number, is set to "1", indicating that the packet has arrived. The out of order delivering problem is solved by assigning

a sequence number to each packet.

7. Negative Acknowledgment and Timeout

   EM applies *negative acknowledgment and timeout* to deal with loss of packets. On the object receiver's side, EM puts a timeout stamp for each shared object being accepted from the network. Once the time period has exceeded the timeout, EM sends a retransmission request to the object sender containing the missing packets and the object name, then it resets the timeout. At the object sender's side, upon receiving the retransmission request, EM will retransfer the required packets for the object again. This procedure continues until all the object packets have been received. Then the receiver acknowledges the object depositor, loads the leased object, and frees the memory. At the other end, upon receiving the acknowledgment for the shared OML object, EM gets ready to send the next object in the ready list.

   EM drops duplicate data packets.

4.6.4 Shared Object Behaviors

As mentioned in section 4.2.3, users can define object behaviors for each object. While parsing this information, the EM parser also stores them. When processing object requests, EM checks whether the object to be sent out has object behaviors and, if it has, in addition to the object code, EM also transfers the object behaviors in the EM description file to the object requester.

After the receiver gets the shared object behaviors, the EM parser is activated to parse the description code and loads in the behaviors for the shared object. When all the packets for the shared object have been received, its instance information stored by the EM parser (see section 4.6.3, step 1) is parsed again to actually generate the instances for the shared object and invoke their behaviors.

Chapter 5

Demonstrations

Three demonstrations built by the Environment Manager (EM) are introduced in this chapter: (1) skeet shooting game, (2) tank battle, and (3) the East Edmonton Mall design environment. Each demonstration shows different design aspects of EM and is representative of a particular application area.

5.1 Skeet Shooting Game

5.1.1 Scenario

This program allows networked players to shoot skeet one at a time. Each player stands at a fixed position and the targets move in such a way as to simulate the angle of flight of a flushed game bird. After each player shoots once, that player determines the following player's target path. All players watch the performance of their competitors. The player with the most hits is the winner.

5.1.2 Implementation

The *simulation token passing* concurrency control is used in this demo. Each player (peer) sits on one machine connected to all the others through the network. Each player's EM world has the same object database.

The right hand is drawn in the environment for each player to represent his/her

position. A shot is triggered by pushing one of the six buttons mentioned in section 4.2.3, then a shooting line is projected out from the player's hand. If this line hits the target and the player has the simulation ownership token, the target will blow up and the player will get one point.

The color of the target indicates which player has the right to shoot the target; in other words, which player has the *simulation ownership token.* The player who sees the target as *yellow* is the *owner* of the simulation (we call him/her the *"owning"* player), and must attempt to shoot the target in order to collect points. After the owning player presses the button and shoots, the target will blow up if hit, and the target for the next *simulation owner* is decided and thrown out. At the same time the new target in the old owner's view changes its color to black, and the next peer on the peer list becomes the new owner and starts collecting points. The new owner sees the target as *yellow*, while all the other players can view the same target, but as *black.* If the owning player blows up the target, not only he/she can see the explosion, but also all the other players. A player can shoot only once before the simulation ownership token is passed along, so a player has a vested interest in aiming and shooting at the target. In effect, *simulation ownership* and *scoring capabilities* are passed from player to player in a *round-robin* fashion.

While the owning player is simulating, it updates the other players' view by sending out shared information of (1) who is the simulation owner, (2) shooting result – whether he/she blows the target up or not, and (3) the target choice for the next owning player. This shared information is represented by three shared variables in an instance called "shared_body" (see appendix B). In section 4.4, we mentioned that when EM receives a shared variable, it appends the sender's host id and process id to the instance name of that shared variable (every shared variable is self-contained; see section 4.6.1) in order to avoid instance name conflict in the receiver's instance table. However for this scenario, rather than sending instances around, the shared variables in an instance called "shared_body" are sent. Therefore we do want every EM world to have an instance whose name is "shared_body", so that all the shared

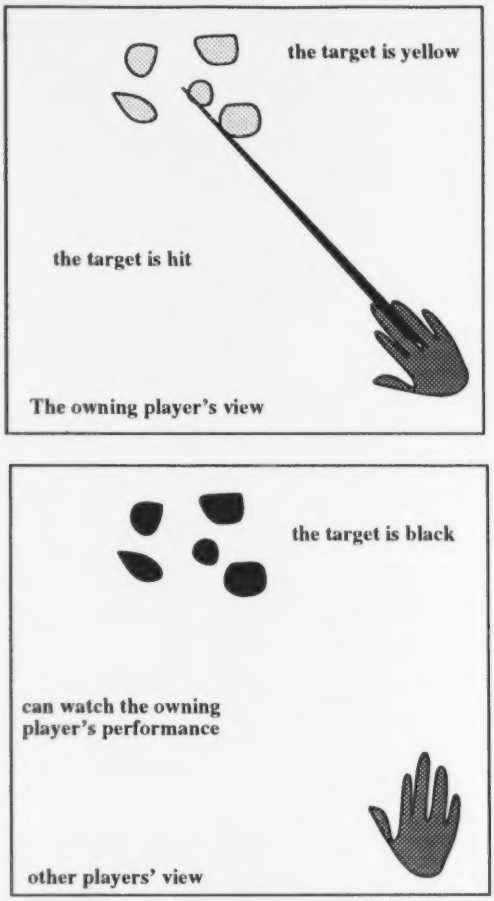


Figure 5.1: A Snapshot Of The Multi-player Skeet Shooting Game

information can be updated in the right instance. EM can turn off the feature of appending the sender's host id and the process id to the instance name by defining the instance to be "*common*". No shared instance is expected to be accepted in this simulation.

In this application, every shared variable has both *readable* and *writable* permissions, no matter which EM world it belongs to. In order to prevent an inconsistent instance state when EM worlds simultaneously try to update it, the *simulation ownership token passing* concurrency control scheme is used here. It maintains the consistency of the distributed virtual world database by restricting manipulation in such a way that only one site at a time can perform operations to alter the status

of the virtual world. In this application, the shared variable "*simulation_owner*" in instance "*shared_body*" represents the token. The EM world whose variable "*simulation_owner*" in instance "*shared_body*" has value "1", can take effective actions, alter the status and broadcast the updates to the other EM worlds.

Figure 5.1 shows the owning player (the one holding the simulation ownership token) attempting to shoot, and the other players, have the same view and can watch the owning player's performance. In other words, not only the owning player's target path but also whether he/she hits the target are shown to the other players.

5.2 Tank Battle

This demonstration is a simplified implementation of SIMNET. Its networked configuration is shown in Figure 4.3.

5.2.1 Scenario

This program enables multiple players/soldiers to participate in a tank battle. The battle-field consists of a large area of ground with hills where tanks can hide. Each soldier can decide the color and the speed of his tank, he can drive his tank, he can hide behind the hill; or he can shoot his opponent tanks. Colliding with a hill, colliding with another tank or being shot by another tank all cause the destruction of the tank. In each soldier's view, he is able to see the activity of every other participant's tank.

5.2.2 Implementation

In this application, each EM world represents a soldier owning a tank and his viewscope. The object space is the same across all the participants. Every soldier can enter and leave the battle simulation at any time. Every EM world broadcasts an instance (*a ghost instance*) representing its soldier's owning tank (*local tank*, defined in section 4.2.2) to all the other EM worlds, so that each soldier can have a view

of all the tanks which are currently in the battle. Each soldier can drive the local tank, change the local tank's heading/direction, stop – to inspect and shoot at his opponent tank. If a tank is destroyed in any EM world, its representations in all other EM worlds can be seen to blow up.

The shared information in this application is specified in the description file: (see appendix C)

- Shared instance

  Each EM world which is the owner of a tank broadcasts an instance representing its tank (local tank) to all other EM worlds. We call this broadcast instance the *ghost instance* of the actual local tank. The ghost instance information is a part of the EM description code containing the object name, instance name, parameters, behaviors and shared variables.

- Expected instances

  In every EM world, a received instance can be accepted and displayed only if it is an instance of object "Tank". As a result, received instances, of any other object, are discarded.

- Shared Variables in the instance of the "Tank" object

  - "heading" represents the direction of the tank.
  - "hit" is a flag indicating whether the tank is destroyed.
  - "tx" and "ty" represent the tank's position (X-axis and Y-axis components),
  - "start_marching" is a flag indicating whether the tank is moving.

  All these variables except "hit" have only *readable* permissions (their values can be sent out) if they belong to the local tank instance, whereas to the ghost tank instances, they only have *writable* permissions (they can be updated). This access restriction enables the owner of the tank to change the tank's

position and heading, and to broadcast the updates. The Ghost tank instances can be updated, but they can't change the values of the actual local tank in its owner's EM world. The variable "hit" is *writable* and *readable* in every instance for object "Tank". A dead-reckoning algorithm is used to extrapolate "tx" and "ty".

Every EM world broadcasts its local tank's ghost instance to all other EM worlds and communicates with the ghost instances in the other EM worlds via shared variables. In the local tank's EM world, the tank's movement and activities are controlled by the soldier using trackers and buttons, whereas in the tank's ghost instance's EM world, the tank's position is simulated using a dead-reckoning algorithm. The ghost tank's heading, the state of being hit and the movement state are updated by the actual local tank's EM world. When an EM world leaves the simulation, all its local tank's ghost instances will be deleted in their respective EM worlds.

The EM world which is the owner of the local tank doesn't need to broadcast the tank's position at every time step; instead, it only needs to inform the tank's ghost instances that their actual local tank is moving. When a ghost tank is updated to the "start moving" state, it begins self-simulating the movement. It is the responsibility of the actual local tank's EM world to generate updates before discrepancies become too large. For this application, in the local tank's EM world, whenever the change of the tank's heading is larger than a threshold, the local tank's current heading, position and moving flag are broadcast.

There is no access restriction on the shared variable "hit" (it is always *readable and writable* in every world). As mentioned in section 4.3.2, this is because if a tank is hit in an EM world, it should be treated as destroyed, no matter whether it has multiple hits in other EM worlds at the same time. That is to say, the variable "hit" can be changed simultaneously by several EM worlds.

Figure 5.2 shows the tank battle scene for each of the two participants. The lower picture shows the view of the soldier driving a black tank. A red tank is in his view. His black tank is blown up by the red tank – you can see the broken muzzle of the

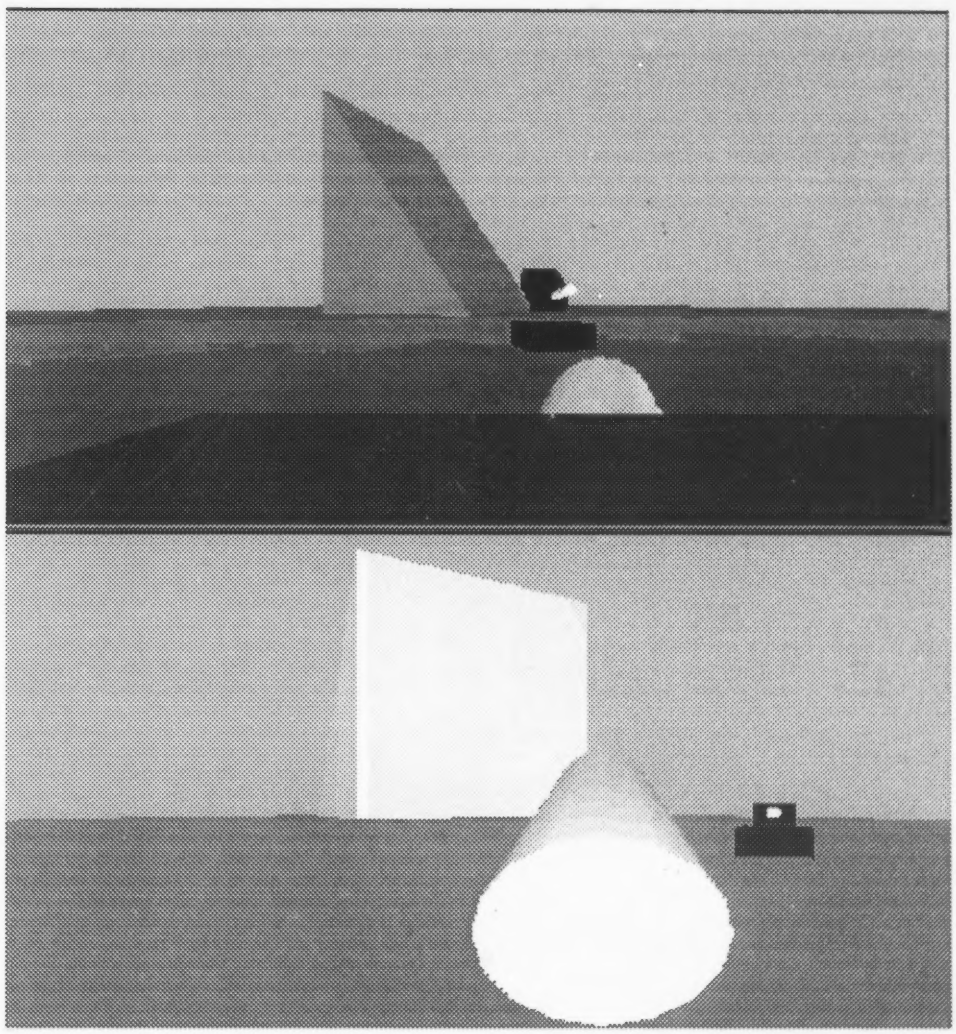


Figure 5.2: A Snapshot Of The Multi-player Tank Battle Simulation

black tank. The upper picture is the view of the soldier driving the red tank; he discovered the black tank and destroyed it.

5.3 The East Edmonton Mall Design Environment

This demonstration presents a simplified environment of the East Edmonton Mall design studio and is representative of networked design systems. Its network configuration is shown in Figure 4.4.

5.3.1 Scenario

This environment is a prototype (representative) of networked design systems that can be supported by EM. Different from the previous two "same-content" networked VR worlds, this environment consists of three EM worlds, each of which has different objects and content. The mall design studio designs environment configurations for both shopping areas and resting areas. Placing plane models and statues in a shopping mall is a very interesting idea. But the mall design studio doesn't design statues, neither does it have plane models. The networked design systems solve this problem by sharing designed pieces (objects) with other studios and museums.

In this networked design system, the mall design studio sends out a request to the network for the specified plane model and statue. The Plane Museum and the Statues Design Studio, which are the other two EM worlds, can deposit their objects in the network upon receiving the request. The mall design studio leases the model and the statue, places them in desired spots and shows their dynamic features (behaviors). Updates for these shared pieces are sent out by their owners whenever any new design idea has been applied.

5.3.2 Implementation

This application is a multi-user-different-content application. Three different EM worlds representing the East Edmonton Mall Design Studio, the Plane Museum and the Statues Design Studio are built at different workstations individually. The mall design studio needs to have a Biplane model and a statue called "Smiling" to put in the mall resting area. The plane museum needs to have a statue called "Life" to put in the hallway. The user in each environment can walk through the environment to see how things look.

The shared information in this demonstration is (see appendix D):

- Shared objects

  An EM world first broadcasts object requests, if it needs to lease some from the network. So the mall design studio EM world broadcasts requests for the objects "Biplane_OBJ" and "Statue_Smiling_OBJ". The Plane Museum EM world requests object "Statue_Life_OBJ" from the network. These requests are sent to the network and to every newly connected EM world until all the requested objects are obtained.

  The Plane Museum EM world, which has the requested object "Biplane_OBJ", sends the object in the form of its OML code, packet by packet, to the mall design studio EM world after receiving the object request. The Statues Design Studio EM world, transfers OML code for object "Statue_Smiling_OBJ" to the mall design studio EM world, and transfers object "Statue_Life_OBJ" to the Plane Museum EM world, after processing the object requests.

  For each EM world which has leased shared objects from the network, the EM world creates one or several instances for the leased objects with the position parameters which are suitable for its own environment. Also, the EM world can accept shared instances of the shared objects from the network.

  In each shared object's owning EM world, a list of all the EM worlds which have leased that object is maintained, so that object updates are sent only to

those that have the affected object and have expressed interest in receiving updates. The object updates are also sent in the form of OML code that the receiving EM world executes to affect the changes. This is possible due to the interpreted nature of OML.

- Shared object behaviors

  Some special features of objects are presented when behaviors are executed. For example, the statue "Life" can sometimes act as a water fountain, and the "Biplane" model can fly along a special path. These behaviors are defined in the objects rather than in the instances.

  Before an EM world deposits an object to the network, it first checks whether the object has behaviors stored by the EM parser. If it has, the behaviors are sent out first in the form of EM description code. In the Statues Design Studio EM world, for example, the "issuing_water" behavior for object "Statue_Life_OBJ", as well as the OML code for object "Statue_Life_OBJ", is sent out to the Plane Museum EM world. Upon receiving shared object behaviors, EM activates its parser to parse and store the information so that the instances created for the shared object can inherit these behaviors. The same procedure is involved in the Plane Museum EM world when depositing object "Biplane_OBJ".

Figures 5.3 to 5.5 show the scenes of three EM worlds involved in the East Edmonton Mall Design Environment.

In Figure 5.3, an instance of shared object "Statue_Life_OBJ" is created and object behavior "issuing_water" is inherited. In Figure 5.4, two statue pieces (statues "life" and "smiling") are shown on the studio shelf. In Figure 5.5, the biplane model and statue "smiling" are leased from the plane museum and the statues design studio, and put into two resting areas. Shared object behavior "fly" and "rotating" are inherited.

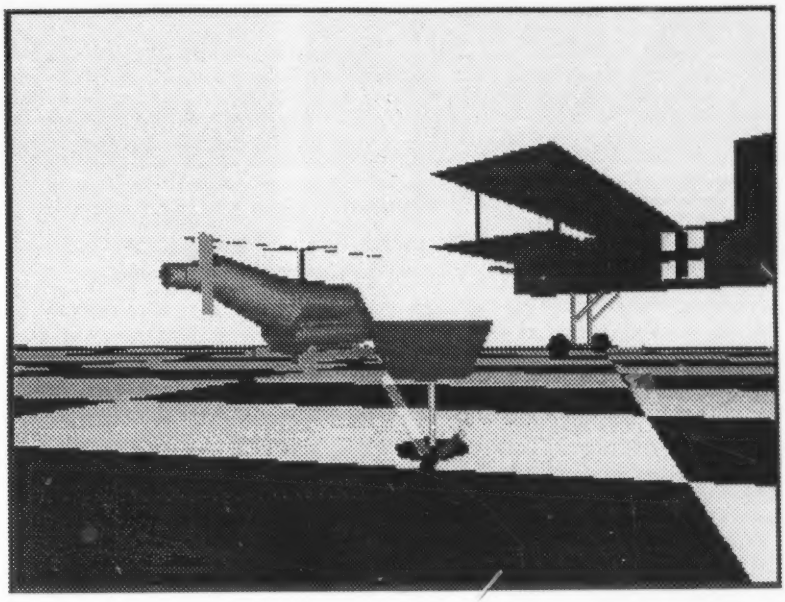


Figure 5.3: A Snapshot Of The Plane Museum Museum Environment

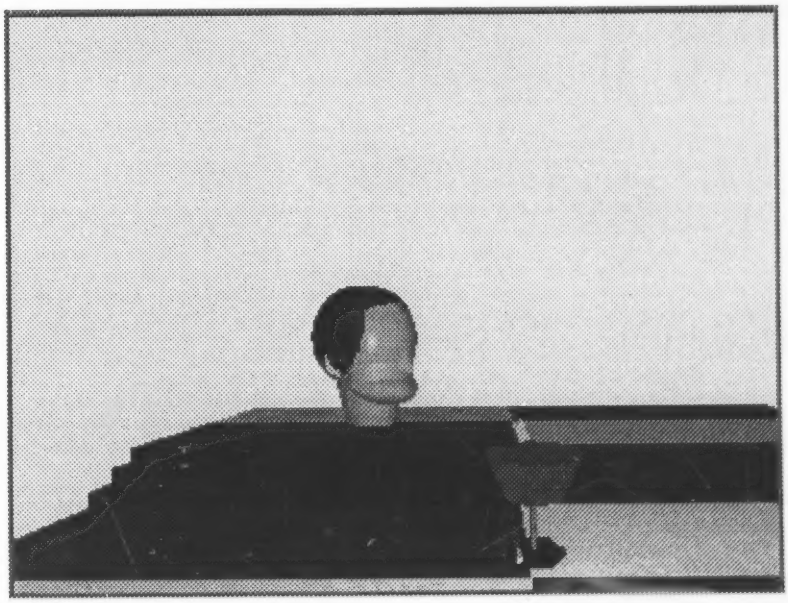


Figure 5.4: A Snapshot Of The Statues Design Studio Environment

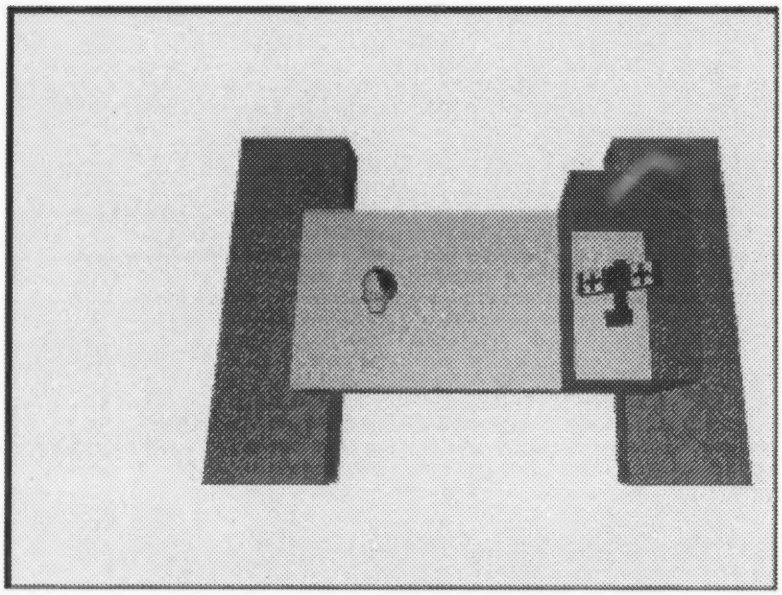


Figure 5.5: A Topview Of The Mall Design Studio Environment

Chapter 6

Conclusions And Future Work

6.1 Conclusions

This thesis work examined issues related to Networked Virtual Reality, such as the system architecture, network configuration, communication protocol, concurrency control and network bandwidth. The Environment Manager (EM), which is a new component of our software tool package for building Virtual Environments, has been implemented with extensions for developing Networked Virtual Environments.

The Environment Manager (EM), Object Modeling Language (OML) and the Minimal Reality (MR) Toolkit provide a rich set of functionalities geared towards expediting the creation of both single-user and networked, multi-user virtual worlds. They eliminate the need to learn about low level graphics, device handling and network programming. This is achieved by providing higher level support for graphical, behavioral and network modeling of virtual worlds in the tool package.

The MR Toolkit provides support for common VR devices and numerous interaction techniques. OML provides geometry and behavioral modeling for three dimensional objects used in virtual worlds. EM sits on top of OML and MR, allowing for the construction of virtual environments without writing any C or MR code. Instead of asking the developer to start from scratch, as is the case for most of other toolkits, EM generates both single-user and networked virtual worlds using a simple

script file where the device configuration, object and instance information (including parameters, behaviors and networked information) are specified. EM provides a "virtual world shell" which is customized by populating it with objects of interest, modifying its behavioral properties and specifying the instances' network behaviors. This makes it possible for the developer to create networked virtual worlds quickly and easily.

In contrast to the *client-server* arrangement for the creation of networked virtual worlds, EM follows the *complete distribution* strategy which is more complex, but more fail-safe and provides less opportunity for communication bottlenecks to occur. In EM, messages are sent only to virtual worlds that have the affected object or instance and have expressed interest in receiving updates. This strategy is very different from broadcasting messages to all worlds whether the messages are relevant or not. The broadcasting approach leads to a significant amount of redundant network traffic in large-scale VR environments. EM also enables a dead-reckoning algorithm to be used to save network bandwidth. Simulation ownership token passing and access permissions are implemented in EM to provide concurrency control. Networked EM virtual worlds can communicate with each other by sharing instance variables, instances, behaviors and objects.

Overall, EM provides higher level support for the creation of networked virtual environments. It reduces the effort required to design a networked VR by providing the standard facilities required by a wide range of networked virtual worlds. In section 3.3, we summarized the ten major steps needed to build a single-user virtual environment using MR and OML. After EM has been implemented, the ten steps are reduced to only one: *given an EM description file, just run "em" with the file name as one of its command line arguments* (see Appendix A.2).

6.2 Comparison With Other Systems

EM is a general virtual environment construction tool. It is aimed at supporting as many VR systems as possible. EM generates both single-user and networked

virtual worlds using a simple script file where the device configuration, object and instance information (including parameters, behaviors and networked information) are specified. This kind of high level support has not been implemented in any of the other systems.

EM applies SIMNET networking technology (see section 2.3.2) and provides support for the SIMNET architecture. In SIMNET, messages are broadcast to all worlds regardless of whether the messages are relevant or not, and therefore it is very inefficient for some applications. EM takes a more efficient approach by sending out messages only to virtual worlds which have the affected object or instance and have expressed interest in receiving updates. This is also the strategy used in VR-DECK, DIVE and BrickNet.

SIMNET, DIVE and VR-DECK support only multi-user-same-content applications that have the same object space across all the users; whereas EM provides support for both multi-user-same-content and multi-user-different-content applications. In other words, EM enables the sharing of object code. As with BrickNet, this is possible due to the interpreted nature of the object representation language (in EM, it is OML and in BrickNet, it is Starship [SSPN94]).

DIVE uses multi-cast protocols to transmit messages. Due to the fact that a certain amount of system management is required to support multi-casting beyond a local-area network, which is not practical for people without super-user privileges, the MR Toolkit peer package doesn't use multi-casting. However, as it stands, the peer package could be re-implemented using multi-casting without the user being aware of the change. *Locking* is used in DIVE to prevent data inconsistency, EM provides a wider and more flexible concurrency control solution for users to choose. The concurrency control schemes in EM include simulation ownership token passing, access permissions, and instance or variable ownership token passing (see section 4.3.2 and section 6.3.2).

EM has many features that are similar to BrickNet. The major difference between these two tools is the following: EM follows a *complete distribution* strategy, whereas

BrickNet adapts a *client-server message control* arrangement. Actually, both MR Toolkit peer package and EM are trying to get away from the "central box", because when many users are connected, the "bottleneck" and "less fail-safe" problems in the client-server arrangement can significantly slow down the simulation, or even crash the whole system if the server fails. BrickNet [SSPN94] proposed an improved approach by having several distributed and cooperative servers and using a high performance machine as the server. The motivation for having a *client-server message control* architecture is that: graphical update and smoothness in interaction are key requirements in virtual worlds, so it is important for the client to delegate as much communication as possible. However, by the time high performance machines are widely available, the motivation for having a client-server arrangement will be minimized compared to solving the bottleneck and less fail-safe problems caused by the increasingly busy network, and server and communications failures.

Although EM uses the unreliable UDP protocol to transmit data, as opposed to VR-DECK which uses reliable TCP protocol, this approach has its own advantages (see section 6.3.1). After communication reliability is improved (see section 6.3.1), EM will have a more flexible, less lagged and more reliable communications protocol.

6.3 Discussions And Future Work

6.3.1 Improving Reliability

EM communication facilities are based on the MR Toolkit peer package which uses UDP datagram protocol. The advantages and disadvantages of using the UDP protocol are listed below:

- Advantages

    1. Connection management is simple.

        There is no overhead of connection establishment and termination. There are no race conditions when two peers seek connections with each other

simultaneously.

2. Packet communication is one-way.

   Because there is no acknowledgment for the data, packets arrive sooner than reliable TCP data packets. Sending device data by one-way communication can reduce device-centered lag substantially.

3. There is no error in the received packet.

   Communication errors do not result in error returns from data sending calls

4. Multi-cast is based on UDP.

   As mentioned in section 3.4, it would be much better to use the Internet multi-cast facility. Because multi-casting is based on the UDP protocol, in considering future re-implementations for multi-casting, it is more reusable for EM to use the UDP protocol than to use the TCP protocol.

- Disadvantages

  UDP packets may be received out of order, or packets may be lost along the way. There is no flow control for transmitting UDP packets.

There is a potential problem in EM because UDP doesn't have flow control over data transferring. TCP implements the flow control by using slide windows, but at the cost of overhead, such as longer packet processing time, larger buffer space, and connection entry tables. For real-time and highly interactive networked virtual worlds, EM drops packets when a receiver can't process a received data item fast enough to keep up with the speed of packet arrivals.

Two approaches can be taken in order to improve the reliability of the EM communications protocol:

1. Adding a reliable protocol to the peer package.

   The experience of dealing with the loss of simulation ownership (see section 4.3.2) and ensuring that large amounts of OML object code are sent reliably (see sec-

tion 4.6.3) shows that adding acknowledgments for infrequent data is a useful extension to the peer package.

2. Using both TCP and UDP protocols.

   For packets containing device data, packet loss is not necessarily a problem, since device data packets are arriving many times per second. For infrequent data sharing, packet loss may prove to be troublesome. Because both TCP and UDP have their strong and weak aspects when applied to developing networked virtual environments, we may leave the choice to the user. That is, we will provide communications using both protocols and let the user decide which protocol to choose for each shared data item. But we may still run into the N-squared connection problem mentioned in section 3.4. It would be much better to use the Internet multi-cast facility after the system management barrier is removed.

6.3.2 Improving Concurrency Control

At present, concurrency control in EM is implemented by using simulation ownership token passing and shared variable permissions. To allow high responsiveness, each EM world has its own copy of the simulation including the shared information. When an operation is requested (e.g. a button press causing an instance to move), the copy locally performs the operation immediately. But this may cause an inconsistent instance state if, at the same moment another EM world is operating on the same instance and its result hasn't been sent out.

One way to solve this problem is to use a technique called *operation transformations* [EG89]. Operation transformation allows high responsiveness by executing local operations immediately. After the execution, the VR world sends out the operation, along with a *state vector* indicating how many operations it has recently processed from other VR worlds. Each VR world has its own state vector, which it compares to incoming state vectors. If the received and local state vectors are equal, the receiving

operation is executed as requested; otherwise it is *transformed* before execution.

The specific transformation is dependent on the operation type (e.g., an *add* or *delete*) and on a log of operations already performed. However, it is still a research issue that whether a user immersed in the virtual reality is happy with both *operation transformations* and being told the action he/she has made is ineffective. In reality, some of the actions are irreversible.

Another concurrency control solution is to use instance or variable ownership token passing. An instance or variable can only be modified by the token owning process. This enables a process to manipulate an instance or a variable exclusively in the local database, and then distribute the changes. For this solution, unlike the ownership concurrency control in BrickNet [SSPN94] and the distributed locks in DIVE [CH93b], there is always one user who owns the instance or the variable token at any time. The current token owner is broadcast to every other user, so that whenever a user wants to modify an instance or a variable, he/she knows where the request should be sent before the actual operation is performed. The following illustrates the five steps that must be taken in order to modify an instance or an attribute (a shared variable) of an instance, if ownership token passing is used to implement the concurrency control.

- A request is made for the ownership of the instance or the attribute to the current owner.
- If the current owner can release the token, the permission is granted. The ownership token passes to the EM world what made the request. This new owner is broadcast to every other EM world. Then the new owner modifies its local copy of the instance or the attribute. If the current owner cannot release the token, the request is rejected and it is the requester's responsibility to decide what to do next.
- Messages containing the changes are sent out to the EM worlds which show interest.

- Each EM world which shows interest receives the message and updates its replica accordingly.

- The ownership is still held by the new owner, but it can be transfered to whoever else wants it.

Using this approach, the local operation can't be performed immediately; the response time will be slower. Whether EM or OML is preferable in implementing this solution is another issue which still needs some thought.

6.3.3 Extensions For Implementing Spatial Locality

In large-scale virtual worlds, another approach to reducing network bandwidth is to limit interest to a spatial locality. For example, for some applications, there is no point in receiving updates to the visual representation of an instance which is too far away to see or interact with.

As an example, we consider a large-scale networked VR with hundreds of interactive manned platform combat simulators (See Figure 6.1). At a particular viewpoint for instance A (it could be a tank or a plane), we can define a view-cone which contains the volume of space visible from that viewpoint for a given field of view and viewing distance. Instance A maintains interest in all instances which fall within a volume that surrounds the view-cone, known as its *caching volume*. All the instances which fall within the caching volume should be *aware* of instance A and send their updates to instance A. Awareness between instances need not be mutually symmetrical [BF92]. The caching volume can be an arbitrary volume but will typically be larger than the view-cone to ensure that visual representations are cached before they are actually visible, and to allow some movement of the viewpoint without causing a cache miss. As illustrated in Figure 6.1, even when instance A can't see instance B, they still need to send their updates to each other, otherwise they won't know when they actually run into each other if they both move backwards.

Instance A's actual interest instance list is the result of the intersection of two

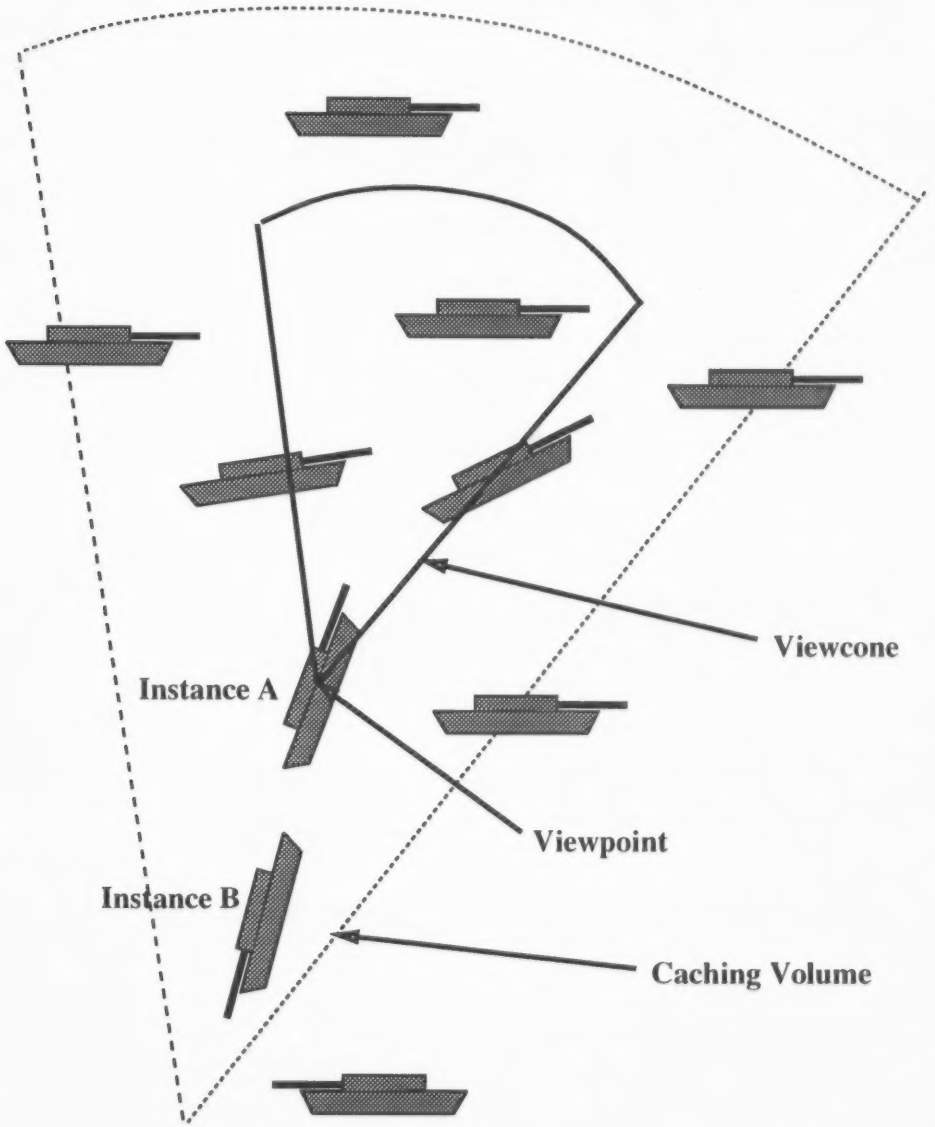


Figure 6.1: A Caching Volume

instance sets. The first set includes all the instances whose caching volume intersects with or contains instance A. The second set contains all the instances which show interest to instance A from the view of the environment's network modeling.

In order to improve the real-time performance of OML, the view volume (view-cone) has already been implemented to cull invisible objects before drawing [Got94]. To implement spatial locality, a good way of determining the caching volume is needed, and this is highly application dependent. However, reducing the network traffic by limiting interest to a spatial locality adds more processing overhead to each EM world. Spatial locality may not fit some applications, in other words, it may not be correct and may affect the result for some applications. EM can make this feature an option for the programmer to choose.

Bibliography

[All91] Earl. A. Alluisi. The Development of Technology for Collective Training: SIMNET, a Case History. *Human Factors 33, 3*, pages 343–362, 1991.

[BF92] Steve Banford and Lennart Fahlen. Aura, Focus and Awareness. *Proceedings of the 5th MultiG Workshop*, 1992.

[BMC92] B. Hughes. Blau, C. E. Moshell, and Lisle. C. Networked Virtual Environment. *ACM SIGGRAPH - 1992 Symposium on Interactive 3D Graphics*, pages 157–160, 1992.

[CDG93] James Calvin, Alan Dickens, and Bob Gaines. The SIMNET Virtual World Architecture. *IEEE Virtual Reality Annual International Symposium'93*, pages 450–455, 1993.

[CH93a] Christer Carlsson and Olof Hagsand. DIVE - a Multi-User Virtual Reality System. *IEEE Virtual Reality Annual International Symposium'93*, pages 394–400, 1993.

[CH93b] Christer Carlsson and Olof Hagsand. DIVE - a Platform For Multi-user Virtual Environment. *Computer and Graphics*, pages 663 – 669, 1993.

[CJKL93] Christopher. F. Codella, Reza Jalili, Lawrence Koved, and J. Bryan Lewis. A Toolkit for Developing Multi-User, Distributed Virtual Environments. *IEEE Virtual Reality Annual International Symposium'93*, pages 401–407, 1993.

[CMe90] Terrence Crowley, Paul Milazzo, and etc. MMConf: An Infrastructure for Building Shared Multimedia Applications. *Proceedings of the ACM 1990 Conference on Computer Supported Cooperative Work*, pages 329–342, 1990.

[CNT92] Christopher Codella, Alan Norton, and Greg Turk. Interactive Simulation in a Multi-Person Virtual World. *ACM Conference on Human Factors in Computing Systems - CHI'92*, pages 329–334, 1992.

[EG89] C. A. Ellis and S. J. Gibbs. Concurrency Control in Groupware System. *SIGMOD Record*, 18-2:399–407, 1989.

[EGR91] C. A. Ellis, S. J. Gibbs, and G. L. Rein. Groupware: Some Issues and Experiences. *Communications of ACM*, 34(1):39–58, 1991.

[Got94] Annette Gottstein. *MSc Thesis: Uses of Bounding Boxes In the Object Modeling Language*. University of Alberta, 1994.

[Gre91] Saul Greenberg. GroupSketch: A Multi-user Sketchpad for Geographically-distributed Small Groups. *Graphics Interface'91*, pages 207–215, 1991.

[Gre93] Mark Green. *Environment Manager, Version 1.0.* Department of Computing Science, University of Alberta, 1993.

[Gre94] Mark Green. *Object Modeling Language (OML), Version 1.1.* Department of Computing Science, University of Alberta, 1994.

[GSW93] Mark Green, Chris Shaw, and Lloyd White. *Minimal Reality Toolkit, Version 1.3.* Department of Computing Science, University of Alberta, 1993.

[How91] Rheingold Howard. *Virtual Reality.* Simon and Schuster, New York, 1991.

[Kaz93] Rick Kazman. Making WAVES: On the Design of Architectures for Low-end Distributed Virtual Environments. *IEEE Virtual Reality Annual International Symposium'93*, pages 443–449, 1993.

[Loe93] Carl Eugene Loeffler. Networked Virtual Reality. *Proceedings of the 5th MultiG Workshop*, 1993.

[LSG91] J. Liang, C. Shaw, and M. Green. On Temporal-Spatial Realism in the Virtual Reality Environment. *UIST 1991 proceedings*, pages 19–25, 1991.

[Sea93a] Sean Halliday. *JDCAD+ To OML Translator (JOT) User's Guide.* Department of Computing Science, University of Alberta, 1993.

[Sea93b] Sean Halliday. *JDCAD+ User's Manual.* Department of Computing Science, University of Alberta, 1993.

[SFB87] M. Stefik, G. Foster, and D. G. Bobrow. Beyond the Chalkboard: Computer Support For Collaboration and Problem Solving in Meeting. *Communications of the ACM*, 30(1):32–47, 1987.

[SG93a] Chris Shaw and Mark Green. The MR Toolkit Peers Package and Experiment. *IEEE Virtual Reality Annual International Symposium'93*, pages 463–470, 1993.

[SG93b] Hanqiu Sun and Mark Green. The use of relation for motion control in an environment with multiple moving objects. *Proceedings of Graphics Interface'93*, pages 209–218, 1993.

[SGLS93] Chris Shaw, Mark Green, Jiangdong Liang, and Yunqi Sun. Decoupled Simulation in Virtual Reality with the MR Toolkit. *ACM Transactions on Information Systems*, pages 287–317, 1993.

[Sha93] Chris Shaw. *The MR Toolkit Peer Package, Version 1.0.* Department of Computing Science, University of Alberta, 1993.

[SLA92] Downes Martin Stephen, Mark Long, and Joanna Alexander. Virtual Reality as a Tool for Cross-cultural Communication: An Example from Military Team Training. *Proceedings of SPIE - The International Society for Optical, conference title: Visual Data Interpretation*, pages 28–38, 1992.

[SSPN94] Gurminder Singh, Luis Serra, Willie Png, and Hern Ng. BrickNet: A Software Toolkit for Network-Based Virtual Worlds. *To appear in: PRESENCE : Teleoperators and Virtual Environments*, MIT Press, MA 1994.

[Tan91] J. C. Tang. Findings from observational studies of collaborative work. *International Journal of Man Machine Studies*, 34(2):In the special edition of CSCW and Groupware, 1991.

[TK92] Haruo Takemura and Fumio Kishino. Cooperative Work Environment Using Virtual Workspace. *Proceedings of the ACM 1992 Conference on Computer Supported Cooperative Work*, pages 226–232, 1992.

[ZPW92] M. J. Zyda, D. R. Partt, and K. P. Wilson. NPSNET: Constructing a 3D Virtual World. *ACM SIGGRAPH - 1992 Symposium on Interactive 3D Graphics*, pages 147–156, 1992.

Appendix A

Environment Manager(EM) - User Manual

A.1 Invoking The Environment Manager

EM is invoked by typing "em" followed by a parameter list at the command line. A typical invocation of EM is:

em *script_file mode control_host*

Each of the command line parameters is described as follows:

- script_file

  This parameter is the name of the script file that describes the environment. This is the only parameter that must be supplied when EM is called. If this parameter has a path specification in addition to a file name, this path is used as the directory where the files mentioned in the script file are located.

- mode

  This parameter is the operating mode for the environment. There are three possible operating modes.

  The **single** mode is used when a single process generating a single image is used for the environment. This is the normal operating mode when debugging an environment.

  The **stereo** mode is used when EM is used to drive a head-mounted display. In this mode two processes are used to run the environment, one for each eye. The specification of this mode consists of the keyboard "stereo" followed by a ":" and the name of the other workstation that the environment runs on. The current workstation is used to generate one eye, and the workstation specified in the mode parameter is used to produce another eye.

  The **shared** mode is used when building a networked virtual environment, in this case the environment is shared by several users. The specification of this parameter consists of the keyword "shared" followed by a ":" and the name of the workstation that coordinates the environment.

CANADIAN THESES

THÈSES CANADIENNES

QUNJIE, WANG

NETWORKED VIRTUAL REALITY.

ISBN

0-315-95099-4

- control_host

  This parameter is the name of the machine that is used to run the control panel. The control panel is used to control some aspects of the environment execution and is used in the debugging process.

Some of the examples of the invocation of the environment manager are:

em *desc*

In this case, the environment manager is invoked with the script file "desc". It will run in the *single* mode by default and will have no control panel.

em *desc* **single** *grassland*

This is the same as the previous example, except that a control panel for the environment will be started on "grassland".

em stereo:*grassland*

This invocation will produce two processes, one running on the current workstation and the other running on "grassland". These two workstations will produce the two images for a head-mounted display.

A.2 The EM Description File Grammar

descfile: header
em_devices
em_share_information
em_object_files_list
em_object_list
em_instance_list
ender;

header: **world** em_world_name |[1];

ender: **end** | ;

em_world_name: Name;

em_devices: **devices**
glove

[1] Yacc grammar, means "or"

```
        end
        |
          devices
          tracker
        end
        | ;

em_share_information : simulation shared information
          round_robin_mode
          leased_objects
          out_going_instances
          expected_objects
          end
        |;


round_robin_mode : round_robin on
        |
          round_robin off
        |;


leased_objects : objects required
          leased_objects_list
        end
        |;


leased_objects_list: one_obj leased_objects_list
        | one_obj
        |;


expected_objects : objects expected
          expected_objects_list
        end
        |;


expected_objects_list: one_obj expected_objects_list
        | one_obj
        |;
```

```
one_obj: Name;

out_going_instances : broadcast_instances
          em_instance_list
         end
         | ;


em_object_files_list : object_files
          objfile_names
         end;


objfile_names : one_obj_file objfile_names
         | one_obj_file;


one_obj_file : Name


em_object_list : one_object em_object_list
         | one_object ;


one_object : object Name
         actions;


actions : actions
         behaviors
         end;


behaviors : one_behavior behaviors
         | one_behavior | ;


one_behavior : event behavior_name
         | condition behavior_name;


event : event_name
         | start '(' event_name ')'
         | middle '(' event_name ')'
```

```
        | stop '(' event_name ')'
        | collision '(' Oml_Instance_Name ')'
        | collision '(' '~' Oml_Instance_Name ')'
        | collision '(' Oml_object_Name ')'
        | collision '(' '~' Oml_object_Name ')'
        | collision '(' ')' ;

event_name : Name | event;

Oml_Instance_Name : Name;

Oml_Object_Name : Name;

behavior_name : Name;

condition : Name;

em_instance_list : em_one_instance em_instance_list
        | em_one_instance
        | ;

em_one_instance : instance em_object_name em_instance_name
          par_list
          actions
          inst_share_information
        |
          common instance em_object_name em_instance_name
          par_list
          actions
          inst_share_information
        |;

inst_share_information: shared information
        shared_variables
        end
        |;
```

```
shared_variables : one_variable
        | one_variable shared_variables ;


one_variable : Name
        type controls;


type : integer | double | color
        | point2 | point3;


controls : control
        | control controls;


control : readable | writable
        | dead_reckoning |;


par_list : '('
        parlist ')'
        | ;


parlist : one_par
        | one_par ',' parlist;


one_par : par
        | Name '=' par ;


par : Integer
    | Double
    | Point2
    | Point3
    | Color
    | String ;


Integer : Unsigned
```

```
    | '-' Unsigned;

Double : DOUBLE
    | '-' DOUBLE

Double_number : DOUBLE
    | '-' DOUBLE

Point2 : '[' Double_number ',' Double_number ']'

Point3 : '[' Double_number ',' Double_number ',' Double_number ']'

Color : '(' Double_number ',' Double_number ',' Double_number ')'
```

Appendix B

An EM Description File For A Skeet Shooting Game

```
world shooting_game

devices
        glove
end

simulation shared information
        round_robin on
end

object_files
        board.oml.cpp.obj
        body.obj
        shooting_line.oml.obj
        block.oml.obj
        cross.oml.obj
        em_share.oml.obj
end

instance test_body body
actions
        tick fix_position
end

common instance EM_share_body share_body
shared information
        choice double readable writable
        simulation_owner integer readable writable
```

```
        to_explode integer readable writable
end

instance shooting_line line
actions
        tick shooting
        tick wait_to_shoot
end

instance block_1_OBJ block (-20.0, 4.0, -4.0)
actions
        collision ( line ) got_it
        to_explode explosion
        target_two_go send_out_target2
        target_four_go send_out_target4
        target_six_go send_out_target6
        need_to_hide hide
        tick set_color
end

instance cross_v_u_OBJ cross (-15.0, 4.0, 9.0)
actions
        collision ( line ) got_it
        to_explode explosion
        target_one_go send_out_target1
        target_three_go send_out_target3
        target_five_go send_out_target5
        need_to_hide hide
        tick set_color
end

instance upper_board_OBJ upper_board_1 (-13.0, -8.0, -6.0)
end
```

Appendix C

An EM Description File For A Tank Battle

```
world tank_simulator

    simulation shared information

        round_robin off

        broadcast_instances

            instance Tank tank1 (10.0, 10.0, 0.5, (1.0, 0.0, 0.0), 0)
            actions

                collision ( hill_OBJ ) explosion
                collision ( Bomb ) got_it
                collision ( Tank ) explosion
                need_to_explode explosion
                tick marching

            end
            shared information

                heading double writable
                tx double writable dead_reckoning
                ty double writable dead_reckoning
                hit integer readable writable
                start_marching integer writable

            end

        end
        objects expected

            Tank

        end
```

```
end

    object_files
            world.obj
            tank.oml.obj
            body.obj
            hill.obj
            biplane2.oml.cpp.obj
            bomb.obj
    end

    instance WORLD terrain (-100.0, 100.0, -100.0, 100.0, 30, 30)

    instance test_body body
    actions
            tick navigation
    end

    instance Tank tank1 (10.0, 10.0, 0.5, (1.0, 0.0, 0.0), 1)
    actions
            tick signal_marching
            begin_marching marching
            tick turning_with_hand
            tick signal_stop
            collision ( hill_OBJ ) explosion
            collision ( Bomb ) got_it
            collision ( Tank ) explosion
            need_to_explode explosion
    end
    shared information
            tx double readable
            ty double readable
            hit integer readable writable
            heading double readable
            start_marching integer readable
    end

    instance Bomb bomb1
    actions
            tick stay
            tick start_to_shoot
```

```
        need_to_bomb bombing
        collision ( hill_OBJ ) stop_shoot
        collision ( Tank ) stop_to_shoot
        need_to_stop stop_to_shoot
end

instance hill_OBJ hill_1 (50.0, 50.0, (0.95, 0.33, 0.85) )

instance hill_OBJ hill_2 (40.0, -30.0, (0.0, 1.0, 0.5) )

instance hill_OBJ hill_3 (-80.0, 70.0, (0.85, 1.0, 0.5) )

instance hill_OBJ hill_4 (-65.0, -75.0, (0.35, 0.4, 1.0) )

end
```

Appendix D

EM Description Files For The East Edmonton Mall Design Environment

D.1 An EM Description File For The Mall Design Studio

```
world East_Edmonton_mall

    simulation shared information
      round_robin off
      objects required
            Biplane_OBJ
            Statue_Smiling_OBJ
      end
    end

    object_files
            ground.obj
            mall.oml.cpp.obj
    end

    instance Ground ground2 (30, 30, -50, -80, -2.0, (0.2, 0.2, 0.2), (1.0, 1.0,1.0))

    instance Mall_OBJ mall (-13.0, -15.0, 0.0)

    instance Biplane_OBJ biplane (-13.0, -15.0, 0.0)
```

```
instance Statue_Smiling_OBJ head ( -13.0, 3.0, 0.0)

end
```

D.2 An EM Description File For The Plane Museum Environment

```
world plane_museum

simulation shared information

  round_robin off
  objects required
                Statue_Life_OBJ
  end

end

object_files
        ground.obj
        biplane2.oml.cpp.obj
        copter.oml.cpp.obj
end

object Biplane_OBJ
actions
        tick fly
end

object copter_OBJ
actions
        tick fly
end

instance Ground ground2 (30, 30, -50, -80, -2.0, (0.2, 0.2, 0.2), (1.0, 1.0,1.0))

instance Biplane_OBJ biplane ( -15.0, 10.0, 2.0)

instance copter_OBJ copter (-15.0, -2.0, 1.0)
```

```
instance Statue_Life_OBJ wine_cup (-6.0, 6.0, -2.0)

end
```

D.3 An EM Description File For The Statues Studio Environment

```
world status_studio

simulation shared information
        round_robin off
end

object_files
        ground.obj
        stair.obj
        head.oml.cpp.obj
        wine_smoke.oml.cpp.obj
end

object Statue_Smiling_OBJ
actions
        tick rotating
end

object Statue_Life_OBJ
actions
        tick issuing_water
end

instance Ground ground1 (2, 2, -10, -30, -1.0, (0.7, 0.0, 0.0), (1.0, 1.0, 1.0))

instance Ground ground2 (2, 2, 2, -30, 0.5, (1.0, 1.0, 1.0), (0.8, 0.0,0.0))

instance Stair stairs (30.0, 0.5, 1.0, 2, 0.0, -15.0, -1.0)

instance Statue_Smiling_OBJ head (7.0, -8.0, 1.5)
```

```
instance Statue_Life_OBJ wine_cup (-6.0, -15.0, 0.0)

end
```